目錄

《精通比特币》第二版

> 来源：tianmingyun/MasterBitcoin2CN

郎咸平说过：比特币白给我都不要

巴菲特：比特币是泡沫，不是一种能够生产价值的资产

紫色的股：为什么说比特币是典型的泡沫

......

也有人说，区块链是最伟大的发明，堪比互联网。

到底比特币是什么？如何判断？难道我们能做的就是人云亦云？

如何能有自己的独立判断？

每个人都有自己的“全知遮蔽”，就像每个人都看不见自己的后脑勺一样。在自己的视野内，在自己的舒适区，如鱼得水，但是就是这种感觉最容易让自己以为“自己以为的”就是客观事实。

正确的态度是研究搞懂，之后才有资格做判断。

这本书就能帮助您全面了解比特币，而且有助于理解其他数字货币。

本书翻译过程中得到了higer（区块链研究社社长）的支持和鼓励，特此致谢。

本书部分段落内容参考摘录了《精通比特币》知笔墨版本，在此特别声明并致谢。

本书附录1比特币白皮书全文摘自巴比特《比特币白皮书：一种点对点的电子现金系统》，在此特别声明并致谢。

以下朋友对本书做出巨大贡献：

菜菜子：翻译了英文版序言，第二版更新说明，词汇表，附录2交易脚本语言操作符，常量和符号等章节

柴春燕和格林怪物：联合翻译附录隔离见证部分

Robbie_英语翻译：第4章审核校对

吴迪：第5章，第9章审核校对

格林怪物：第6章审核校对

阿龙：第7章，第11章审核校对

阮立志和冯锦炜：第10章审核校对

琳：第12章审核校对

黄豆：封面封底扉页以及其他内容设计

由于时间原因和个人水平能力原因，初稿中有许多格式和理解翻译错误。以上各位朋友在审核校过程中修正了初稿中许多错误，甚至部分章节兼职了翻译工作，在此表示致歉和感谢。

即便如此，当前版本还可能存在部分错误，欢迎读者在github上提交勘误，也可以发至邮箱：yuntianming@aliyun.com

乔延宏 2017.11.11

Mastering Bitcoin - Second Edition 版权说明

The second_edition branch, is the source for the published versions of Mastering Bitcoin (Second Edition).

The tag second_edition_print_1, corresponds to the first print of the second edition.

1.1什么是比特币？

比特币是构成数字货币生态系统基础的概念和技术的集合。称为比特币的货币单位用于存储和传输比特币网络中的参与者之间的价值。比特币用户主要通过互联网使用比特币协议进行通信，尽管也可以使用其他传输网络。可用作开源软件的比特币协议栈可以在各种计算设备（包括笔记本电脑和智能手机）上运行，从而使该技术易于访问。

用户可以通过网络传输比特币，就像常规货币一样方便操作即可完成任何事情，包括买卖商品，汇款给别人或组织，或者扩大信用。比特币可以以专门的货币兑换方式购买，出售和兑换其他货币。比特币在某种意义上是网络的完美形式，因为它是快速，安全和无地域边界的。

与传统货币不同，比特币完全是虚拟的。没有物理硬币，甚至数字货币本身。这种币隐含在从发送方到收件人转移价值的交易中。 比特币用户有自己的密钥，允许他们证明比特币网络中的比特币的所有权。使用这些密钥，他们可以签署交易以解锁价值，并将其转移给新的所有者实现消费。钥匙通常存储在每个用户的计算机或智能手机上的数字钱包中。拥有可以签署交易的密钥是消费比特币的唯一先决条件，通过密钥实现每个用户的完全控制。

比特币是分布式的对等系统。因此，没有“中央”服务器或控制点。比特币是通过称为“挖掘”的过程创建的，该过程涉及到在处理比特币交易时竞争寻找数学问题的解决方案。比特币网络中的任何参与者（即，使用运行完整比特币协议栈的设备的任何人）可以作为矿工使用其计算机的处理能力来验证和记录交易。平均每10分钟，有人可以验证过去10分钟的交易，并获得全新的比特币奖励。基本上，比特币采矿分散了中央银行的货币发行和结算功能，取代了任何中央银行的需求。

比特币协议包括内置的算法，用于调整整个网络的采矿功能。平均而言，任何时候，无论多少矿工（以及多大处理能力）参与竞争，矿工必须执行的处理任务难度实现动态调整，保证每10分钟就可以挖矿成功。该协议还将每4年发行新比特币的比例降低一半，并将将发行的比特币的总数限制在低于2100万硬币的固定总量。结果是，流通中的比特币数量紧随其后的一个容易预测的曲线，到2140年将达到2100万。由于比特币的发行率下降，长期来看，比特币货币是通货紧缩。此外，比特币不能通过“打印”超过预期发行率的新货币来膨胀。

换句话说，比特币（bitcoin）也是协议，对等网络和分布式计算创新的代名词。比特币货币真的只是本发明的第一个应用。

比特币代表了数十年密码学和分布式系统研究的高潮，包括四个关键创新，这四个创新以独特和强大的组合结合在一起。

比特币这四个创新包括：

- 去中心化的对等网络（比特币协议）

- 公共交易总帐（区块链）
- 独立交易确认和货币发行的一套规则（共识规则）
- 实现有效的区块链全球去中心化共识的机制（工作量证明算法）

作为一名开发人员，我将比特币视为货币互联网，通过分布式计算传播价值和确保数字资产所有权的网络。比特币还有很多比起第一眼看到的更多的内容。在本章中，我们将介绍一些主要的概念和术语，获得必要的软件，并使用比特币进行简单的交易。在接下来的章节中，我们将开始展开使比特币成为可能的技术层次，并检查比特币网络和协议的内部工作。

比特币之前的数字货币可行的数字货币的出现与密码学的发展密切相关。真正的挑战是当使用比特来代表可以交换商品和服务的价值却不以为奇。接受数字金钱的人的三个基本问题是：

我可以相信钱是真实的，不是假的吗？

我可以相信数字金钱只能花一次（被称为“双重支付”）吗？

我可以确定没有人能够声称这笔钱属于他们而不是我吗？

纸币发行商通过使用越来越复杂的纸张和印刷技术不断打击假冒问题。物理钱容易解决双重支付问题，因为同一纸币不能同时在两个地方。当然，传统的钱也经常以数字方式存储和传送。在这些情况下，假冒和双重支出问题是通过中央权威机构清算所有电子交易来处理的，中央权威机构拥有面向全球的货币视角。对于不能利用深奥油墨技术或全息条码的数字货币，密码术为信任用户对价值权利的合法性提供了依据。具体来说，加密数字签名使用户能够签署数字资产或证明该资产所有权的交易。通过适当的架构，数字签名也可用于解决双重支出问题。

当加密开始在20世纪80年代末开始变得更广泛的可用性和理解时，许多研究人员开始尝试使用加密技术构建数字货币。这些早期的数字货币项目发行数字货币，通常由国家货币或贵金属（如黄金）支持。

虽然这些早期的数字货币是有效的，但它们是集中的，因此很容易被政府和黑客攻击。早期的数字货币使用中心化的票据交易所定期进行所有交易，就像传统的银行系统一样。不幸的是，在大多数情况下，这些新兴的数字货币是政府担忧的目标，最终从法律上逐渐消失了。还有些由于当母公司突然清盘就失败了。无论是合法的政府还是犯罪分子，都需要去中心化的数字货币来避免单一的攻击避免对抗者的干预。比特币就是一种这样一个系统，通过设计实现去中心化，并且不受制于任何可能被攻击或损坏的中央权威或控制点。

1.2比特币历史

Bitcoin是在2008年由署名Satoshi Nakamoto的牛人发明的，他出版了一篇题为“Bitcoin：A Peer-to-Peer Electronic Cash System”的文章[1]。 Nakamoto结合了诸如b-money和HashCash等先前的发明，创建了一个完全去中心化的电子现金系统，它不依赖中央机构进行货币发行或结算和验证交易。关键的创新是使用分布式计算系统（称为“工作量证明”算法）每10分钟进行一次全球性的“选举”，从而允许分布式网络达成关于交易状态的共识。这优雅地解决了双重支出的问题，就是一个货币单位可以花费两次。以前，双重支出问题是数字货币的弱点，并通过中心化的票据交换所清算所有交易来解决。

比特币网络始于2009年，基于中本聪发布的参考实施指南，之后由许多其他程序员进行修订。为比特币提供安全性和弹性的工作量证明算法（挖掘）的实施以指数级增长，现在超过了世界顶级超级计算机的组合处理能力。比特币的总市值有时超过200亿美元，这取决于比特币兑美元的汇率。到目前为止，网络处理最大的交易是1.5亿美元，即时传输，无需任何费用处理。

Satoshi Nakamoto于2011年4月退出公众视线，将代码和网络的责任放在一个蓬勃发展的志愿者小组身上。比特币背后的这个人身份仍然未知。然而，中本聪和任何人都没有对比特币系统进行个人控制，这个系统基于完全透明的数学原理，开放源代码和参与者之间的共识持续运行。本发明本身具有开创性，已经延伸到分布式计算，经济学和计量经济学领域。

分布式计算问题的解决方案Satoshi Nakamoto的发明也是分布式计算当中一个古老问题的实用和新颖的解决方案，这就是 “拜占庭式将军”问题。简而言之，这个问题包括通过在不可靠和潜在的妥协网络上交换信息来尝试商定一个行动方案或一个系统的状态。 Satoshi Nakamoto的解决方案使用工作量证明的概念在没有中央信任机构的情况下实现共识，代表了分布式计算的突破，并具有超越货币的广泛适用性。可以用来达成一致的分权网络，比如彩票，资产登记，数字公证等等以证明选举的公平性。

1.3比特币使用，用户和他们的故事

比特币是古老的技术“钱”的创新。其核心在于钱方便了人与人之间的价值交流。因此，为了充分了解比特币及其用途，我们将从使用它的人的角度审视它。这里列出的每个人和他们的故事都说明了一个或多个具体的用例。我们将在整本书中看到他们：

北美低价值零售业Alice住在北加州湾区。她听她的从事技术工作的朋友说过比特币，因此想要开始使用它。我们将跟随她的故事，在她学习比特币，购买一些，然后花费一些她的比特币在帕洛阿尔托的Bob咖啡厅买一杯咖啡时。这个故事将从零售消费者的角度向我们介绍软件，交易所和基本交易。

北美高附加值零售Carol是旧金山的艺术画廊老板。她卖昂贵的绘画换取比特币。这个故事将介绍高价值物品零售商“51%”共识攻击的风险。

离岸合同服务Bob，帕洛阿尔托的咖啡店老板，正在建立一个新的网站。他与印度的网络开发商Gopesh签约，后者在印度班加罗尔居住。 Gopesh同意在比特币中支付。这个故事将研究使用比特币进行外包，合同服务和国际电汇。

网上商店Gabriel是里约热内卢的一个有进取心的年轻青少年，经营着一家小型网店，销售比特币品牌的T恤，咖啡杯和贴纸。Gabriel太年轻，没有银行账户，但他的父母鼓励他的企业精神。

慈善捐款Eugenia是菲律宾儿童慈善机构的主任。最近她已经发现了比特币，并希望利用它来接触新的国内外捐助者，为她的慈善筹款。她还在调查使用比特币快速将资金分配给需要的地区的方法。这个故事将展示使用比特币来跨币种和跨国界的全球筹款活动，以及在慈善组织中使用开放透明的分类账簿。

进出口Mohammed是迪拜的电子进口商。他正在尝试使用比特币从美国和中国购买电子产品，进口到阿联酋，以加速进口付款过程。这个故事将展示如何将比特币用于与物理商品相关的大型企业之间的国际支付。

采矿为比特币Jing是上海的计算机工程专业学生。他已经使用他的工程技术来建立一个"采矿"矿机来挖掘比特币来增加他的收入。这个故事将研究比特币的"工业"基础：用于确保比特币网络和发行新货币的专门设备。

这些故事中的每一个都是基于目前使用比特币的真实人物和实际行业，为全球经济问题创造新的市场，新的行业和创新的解决方案。

1.4入门

比特币是使用同样协议的客户端应用程序访问的协议。"比特币钱包"是比特币系统最常见的用户界面，就像Web浏览器是HTTP协议最常用的用户界面一样。有很多实施和品牌的比特币钱包，就像有许多品牌的网络浏览器（例如，Chrome，Safari，Firefox和Internet Explorer）。就像我们都有我们最喜欢的浏览器（Mozilla Firefox，Yay！）和我们不喜欢的（Internet Explorer，Yuck！），比特币钱包的质量，性能，安全性，隐私和可靠性各不相同。还有一个比特币协议的参考实现，其包括被称为"Satoshi客户端"或"比特币核心"的钱包，该钱包源于由Satoshi Nakamoto撰写的初始客户端。

1.4.1选择比特币钱包

比特币钱包是比特币生态系统中最活跃的开发的应用之一。这里竞争激烈，目前存在可能正在开发新的钱包，但也有去年的几个钱包已不再积极维护。许多钱包专注于特定的平台或具体用途，一些更适合初学者，而其他的钱包则为高级用户提供了功能。选择钱包是非常主观的，取决于使用和用户的专业知识。因此，不可能推荐一个特定的钱包品牌或项目。然而，

我们可以根据其平台和功能对比特币钱包进行分类，并提供所有不同类型的钱包的一些建议。更好的是，在比特币钱包之间移动钱是容易，便宜和快速的，所以值得尝试几种不同的钱包，直到找到符合您需求的钱包。

根据平台，比特币钱包可以分类如下：

桌面钱包桌面钱包是作为参考实现创建的第一种类型的比特币钱包，许多用户运行桌面钱包以实现其功能，自主性和控制权。在通用操作系统（如Windows和Mac OS）上运行具有一定的安全隐患，因为这些平台往往不安全，配置不当。

手机钱包手机钱包是比特币钱包最常见的类型。在智能手机操作系统（如Apple iOS和Android）上运行，这些钱包通常是新用户的绝佳选择。许多都是为了简单易用而设计的，但也有功能强大的用户的全功能移动钱包。

在线钱包Web钱包通过网络浏览器访问，并将用户的钱包存储在由第三方拥有的服务器上。这类似于webmail，因为它完全依赖于第三方服务器。其中一些服务使用在用户浏览器中运行的客户端代码进行操作，该代码可以控制用户手中的比特币密钥。然而，大多数人需要在安全和方便性之间进行妥协。在第三方系统上存储大量的比特币是不合适的。

硬件钱包硬件钱包是在专用硬件上独立操作比特币钱包的设备。它们通过USB与桌面网络浏览器或通过移动设备上的近场通信（NFC）进行操作。通过专用硬件进行所有比特币相关操作，这些钱包被认为是非常安全的，适合存储大量的比特币。

纸钱包控制比特币的密钥也可以打印长期存储。即使可以使用其他材料（木材，金属等），这些也被称为纸钱包。纸钱包提供低技术但高度安全的长期存储比特币的方法。脱机存储也经常被称为冷存储。

对比特币钱包进行分类的另一种方法是通过他们的自主程度以及它们如何与比特币网络进行交互：

全节点客户端完整客户端或“完整节点”是存储比特币交易的全部历史（每个用户每次交易）的客户端，管理用户的钱包，并且可以直接在比特币网络上启动交易。完整节点处理协议的所有方面，并可以独立地验证整个区块链和任何交易。全节点客户端消耗大量计算机资源（例如，超过125 GB的磁盘，2 GB的RAM），但提供完全自主和独立的交易验证。

轻量级客户端一个轻量级的客户端，也称为简单支付验证（SPV）客户端，连接到比特币完整节点（前面提到过的），用于访问比特币交易信息，但是在本地存储用户钱包，并独立地创建，验证和传输交易。轻量级客户端与比特币网络直接交互，无需中介。

第三方API客户端第三方API客户端是通过应用程序编程接口（API）的第三方系统与比特币交互的API客户端，而不是直接连接到比特币网络。 这时钱包可能由用户或第三方服务器存储，但所有交易都需要通过第三方。

结合这些分类，比特币钱包可以分为几个小组，三个最常见的划分是桌面全客户端，移动轻巧钱包和网络第三方钱包。不同类别之间的界限通常是模糊的，许多钱包在多个平台上运行，并且可以以不同的方式与网络进行交互。

为了本书的目的，我们将演示使用各种可下载的比特币客户端，从参考实现（Bitcoin Core）到移动和网络钱包。一些示例将需要使用Bitcoin Core，除了作为完整的客户端，还可以将API暴露给钱包，网络和交易服务。如果您计划探索比特币系统中的编程接口，则需要运行Bitcoin Core或其他客户端（参见[alt_libraries]）。

1.4.2快速开始

我们在比特币使用，用户和他们的故事中介绍的Alice不是技术行家，最近只听到她的朋友Joe提到过比特币。在聚会上，Joe再次热烈地向他周围解释了比特币，并提供演示。有趣的是，Alice问她如何开始使用比特币。Joe说，手机钱包最适合新用户，他推荐了他最喜欢的几款钱包。Alice下载Android的“Mycelium”，并将其安装在手机上。

当Alice首次运行Mycelium时，与许多比特币钱包一样，应用程序会为她自动创建一个新的钱包。Alice在她的屏幕上看到钱包，如“Mycelium手机钱包”如下图1-1所示（注意：不要将比特币发送到此示例地址，它将永远丢失）。

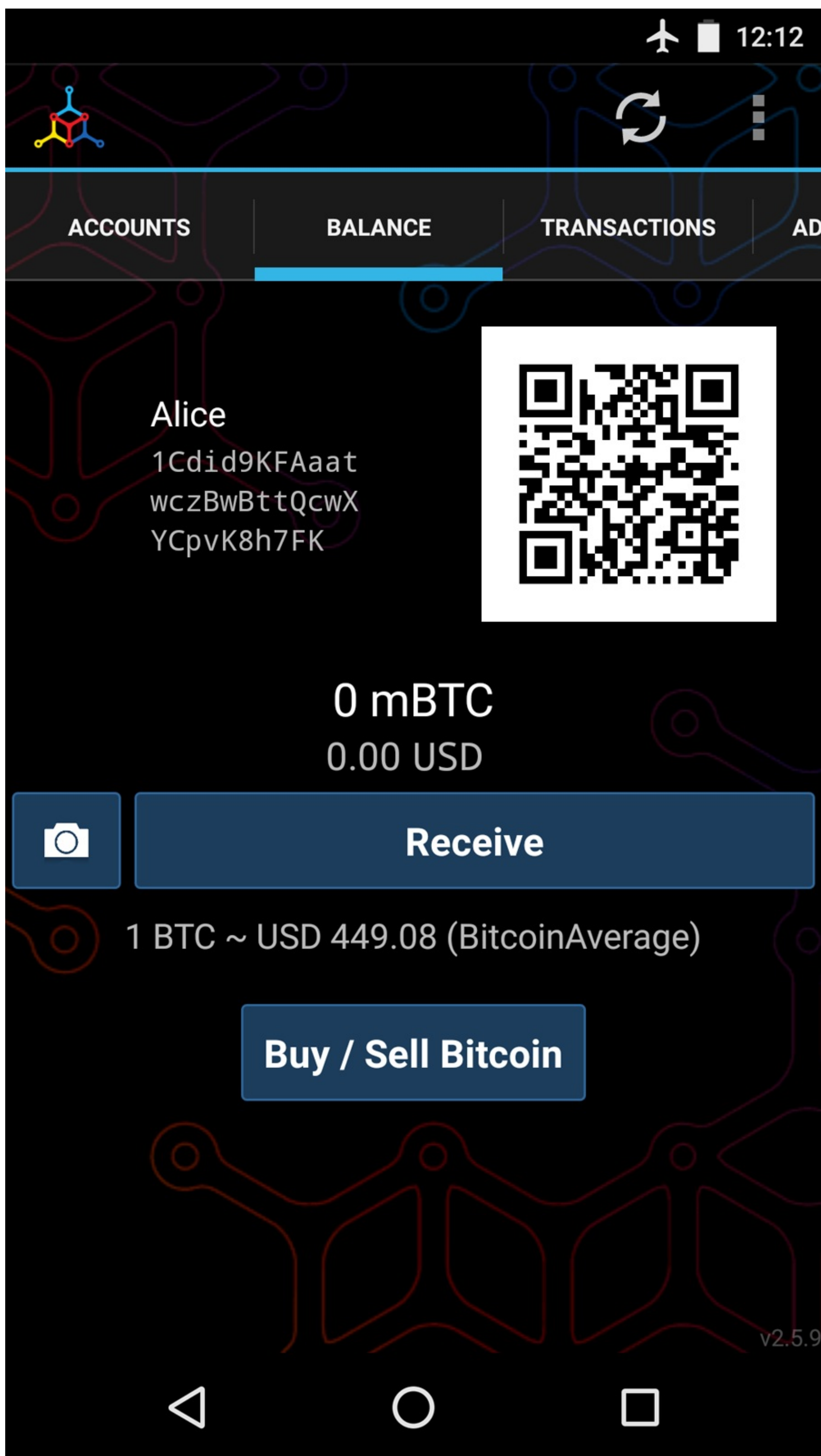

12:12
ACCOUNTS
BALANCE
TRANSACTIONS
AD
Alice
1Cdid9KFAaat
wczBwBttQcwX
YCpvK8h7FK
0 mBTC
0.00 USD
Receive
1 BTC ~ USD 449.08 (BitcoinAverage)
Buy / Sell Bitcoin
v2.5.9

图1-1 Mycelium移动钱包

这个屏幕最重要的部分是Alice的比特币地址。 在屏幕上，它显示为一长串字母和数字：1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK。 钱包的比特币地址旁边是一个二维码，一种形式的条形码，包含可由智能相机扫描的格式的相同信息。二维码是具有黑色和白色点的图案的正方形。 Alice可以通过点击二维码或接收按钮将比特币地址或二维码复制到剪贴板上。 在大多数钱包中，点击二维码也会放大，以便更容易通过智能手机相机进行扫描。

提示比特币地址以1或3开头。像电子邮件地址一样，他们可以与其他可以使用它们的比特币用户共享，直接将比特币发送到您的钱包。 从安全角度看，关于比特币地址没有任何敏感性。 它可以在任何地方发布，而不会危及帐户的安全。 与电子邮件地址不同，您可以随意创建新的地址，所有这些都会将资金用于您的钱包。 事实上，许多现代钱包为每个交易自动创建一个新地址，以最大限度地提高隐私。 钱包只是一个地址的集合和解锁资金的钥匙。

Alice现在准备好收到资金。她的钱包应用程序随机生成一个私钥（在[private_keys]中更详细地描述）及其相应的比特币地址。在这一点上，她的比特币地址对于比特币网络来说是不知道的，或者是未经注册到比特币系统中。她的比特币地址只是一个数字，对应于一个可以用来控制资金访问的密钥。它是由她的钱包独立生成的，还没有参考或注册任何服务。事实上，在大多数钱包中，比特币地址和任何外部可识别的信息（包括用户的身份）之间没有关联。在该地址被引用作为比特币总帐的交易中的接收者之前，比特币地址只是在比特币中有效的大量可能的地址的一部分。只有一旦与交易相关联才能成为网络中已知地址的一部分。

Alice现在可以开始使用她新的比特币钱包了。

1.4.3得到你的第一个比特币

新用户的第一个也是最困难的任务是获取一些比特币。与其他外币不同，您还不能在银行或自助机购买比特币。

比特币交易是不可逆转的。大多数电子支付网络（如信用卡，借记卡，PayPal和银行帐户转帐）都是可逆的。对于销售比特币的人来说，这种差异引起了很高的风险，买方在收到比特币后会扭转电子支付，实际上欺骗了卖家。为了减轻这种风险，接受传统电子支付的公司通常要求买方进行身份验证和信用验证（可能需要几天或几周时间）。作为新用户，这意味着您不能立即使用信用卡购买比特币。需要有一点耐心和创造性的想法，但是不要着急。

以下是作为新用户得到比特币的一些方法：

找一个有比特币的朋友，直接从他或她那里买一些。许多比特币用户都是以这种方式开始的。这种方法是最不复杂的。找到比特币持有者的好办法是参加Meetup.com上列出的本地比特币会议。（在中国根本无需这么麻烦，加微信群，在线支付就可以）

使用分类服务，如localbitcoins.com来查找您所在地区的卖家，以便在现场交易中购买比特币。

通过卖比特币的产品或服务赚取比特币。如果你是程序员，出售你的编程技巧。如果你是美发师，理发只收比特币。

在你的城市使用比特币ATM。比特币自动取款机是接受现金并将比特币发送到智能手机比特币钱包的机器。使用Coin ATM Radar的在线地图找到靠近您的比特币ATM。

使用与您的银行帐户相关联的比特币货币兑换。现在有很多国家都有货币交易所，为买卖双方交易使用当地货币进行交易。实时行情服务（如BitcoinAverage）通常会显示每种货币的比特币交易所列表。

提示比特币与其他支付系统的优点之一是，当正确使用时，它为用户提供了更多的隐私。 获取，持有和支付比特币不要求您向第三方泄露敏感和个人身份信息。 但是，如果比特币涉及传统的货币交换系统，那么国家法律和国际法规就会适用。 为了兑换本币的比特币，您通常需要提供身份证明和银行信息。 用户应该知道，一旦比特币地址附加到一个身份，所有关联的比特币交易也很容易识别和跟踪。 这是许多用户选择维护专用交易账户与其钱包进行分离的一个原因。

Alice听朋友介绍比特币，所以她有一个简单的方法来获得她的第一个比特币。接下来，我们将看看她如何从她的朋友Joe 购买比特币，以Joe 如何将比特币发送到她的钱包。

1.4.4查找比特币当前价格

在Alice可以从Joe 购买比特币之前，他们必须同意比特币和美元之间的汇率。这给新兴的比特币带来了一个共同的问题："谁设定比特币价格？"简单的答案是价格是由市场设定的。

比特币与大多数其他货币一样，有浮动汇率。这意味着比特币相对于任何其他货币的价值都会根据交易的各个市场的供求情况而波动。例如，以美元计算的比特币的"价格"是根据最近的比特币和美元交易在每个市场中计算的。因此，价格往往每秒钟几次波动。定价服务将汇总来自几个市场的价格，并计算代表货币对的广泛市场汇率（例如BTC / USD）的数量加权平均数。

有数百个应用程序和网站可以提供当前的市场利率。这里有一些最受欢迎的：

Bitcoin Average一个网站，提供每种货币的体积加权平均数的简单视图。

CoinCap一项服务列出了数百种加密货币（包括比特币）的市值和汇率。

Chicago Mercantile Exchange Bitcoin Reference Rate可用于机构和合同参考的参考值，作为CME投资数据的一部分提供。

除了这些不同的网站和应用程序，大多数比特币钱包都将自动转换比特币和其他货币之间的兑换数量。 在将比特币发送给Alice之前，Joe将使用自己的钱包自动转换价格。

1.4.5发送和接收比特币

Alice决定将10美元转换成比特币，以免对这种新技术冒太多的险。她给Joe 10美元现金，打开她的Mycelium钱包应用程序，并选择Receive。这将显示一个二维码与Alice的第一个比特币地址。

Joe然后在他的智能手机钱包上选择发送，并显示一个包含两个输入的屏幕：

目的比特币地址以比特币（BTC）或其当地货币（USD）发送的金额

在比特币地址的输入字段中，有一个看起来像二维码的小图标。这样Joe可以用他的智能手机相机来扫描条形码，这样他就不必输入Alice的比特币地址，这需要非常长的时间，而且容易出错。 Joe点击二维码图标并激活智能手机相机，扫描Alice智能手机上显示的二维码。

Joe现在将Alice的比特币地址设置为收件人。Joe输入的金额为10美元，他的钱包通过访问在线服务的最新汇率来转换它。当时的汇率是每个比特币$ 100美元，所以如Joe的钱包（见图1-2Airbitz移动比特币钱包发送屏幕）截图所示，10美元的价值是0.10比特币（BTC）或100毫比银币（mBTC）。

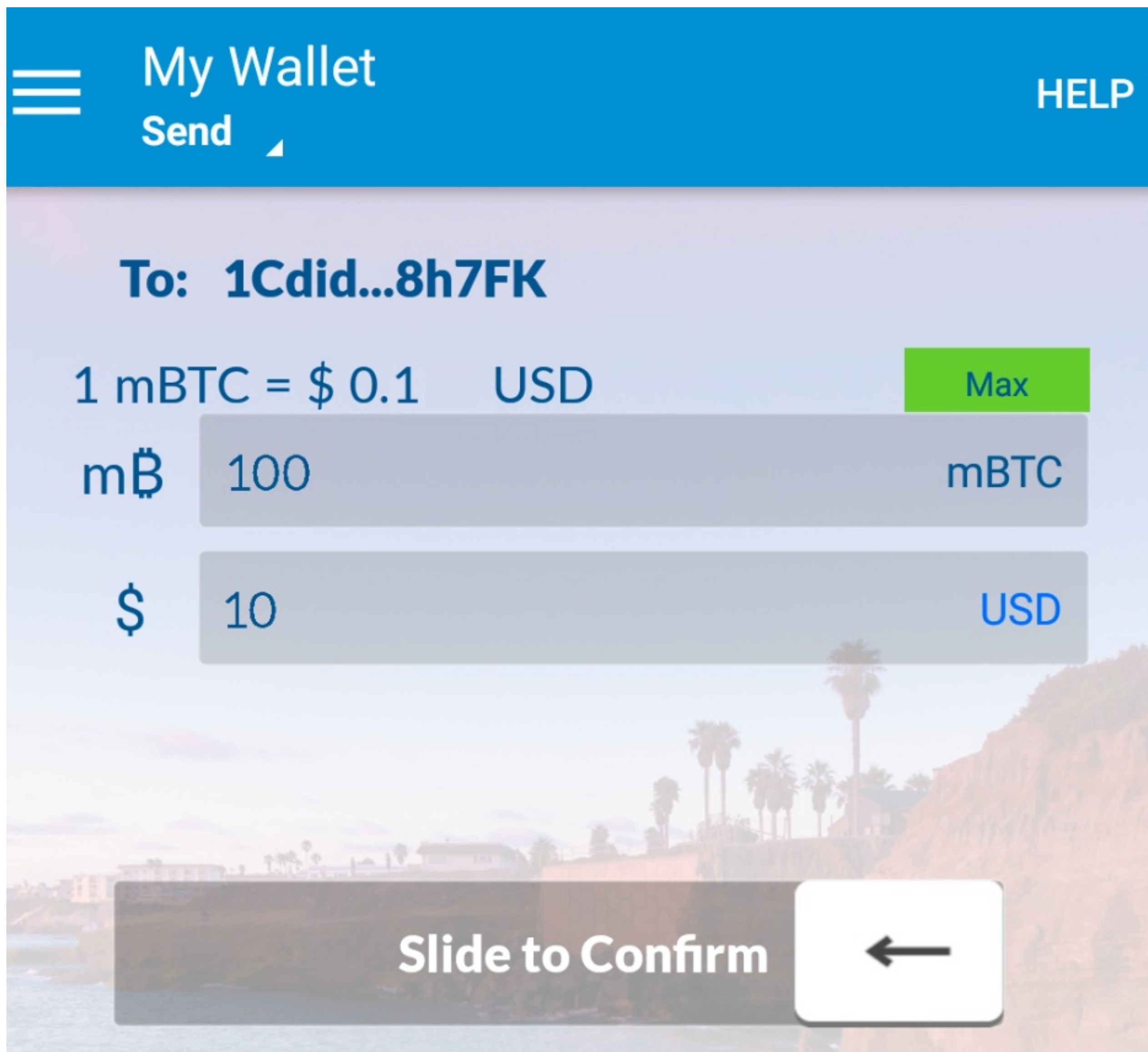


图1-2. Airbitz移动比特币钱包发送屏幕Joe然后仔细检查，以确保他已经输入了正确的金额，因为他要汇款，错误是不可逆转的。经过仔细检查地址和金额后，他按发送方式传送交易。Joe的移动比特币钱包构建了一个交易，从Joe的钱包将0.10 BTC发送给Alice提供的地址，并用Joe的私钥签署交易。这告诉比特币网络，Joe已经授权将这笔钱转移给Alice的新地址。当交易通过对等协议传输时，它会快速传播到比特币网络。在不到一秒钟内，网络中大多数连接良好的节点都会首次接收到交易，并且首次看到Alice的地址。

同时，Alice的钱包不断地“倾听”在比特币网络上发布的交易，寻找与她的钱包中的地址匹配的任何内容。在Joe的钱包发送交易几秒钟后，Alice的钱包将显示它正在接收0.10 BTC。

确认起初，Alice的地址将把Joe的交易显示为“未确认”。这意味着交易已传播到网络，但尚未记录在比特币交易账簿即区块链中。要确认，一个交易必须包含在一个区块中，并被添加到区块链，这样的情况平均每10分钟发生一次。在传统的财务术语中，这被称为清算。有关比特币交易的传播，验证和清算（确认）的详细信息，请参阅挖矿章节[mining]。

Alice现在可以自豪地称自己是0.10 BTC的所有者了，她有权花费这些钱了。在下一章中，我们将首先用比特币进行购买，并更详细地研究底层交易和传播技术。

参考内容：

1. "Bitcoin: A Peer-to-Peer Electronic Cash System," Satoshi Nakamoto (https://bitcoin.org/bitcoin.pdf).

2.1交易，块，挖矿和区块链

比特币系统与传统的银行和支付系统不同，是基于去中心化的信任。在比特币中，取代中央信任机构的是，信任是通过比特币系统中不同参与者的相互作用达成的。 在本章中，我们将通过较高层面跟踪比特币系统单笔交易，观察交易如何通过比特币分布式共识机制变得可信，被接受，并且最终记录在区块链，也就是所有交易的分布式账簿。 随后的章节将深入探讨交易，网络和采矿背后的技术。

2.1.1比特币概述

如图2-1所示的概览图中，我们可以看到比特币系统由用户（用户通过密钥控制钱包）、交易（每一笔交易都会被广播到整个比特币网络）和矿工（通过竞争计算生成在每个节点达成共识的区块链，区块链是一个分布式的公共权威账簿， 包含了比特币网络发生的所有的交易）组成。

本章中的每个示例都基于在比特币网络上进行的实际交易，通过将资金从一个钱包发送到另一个钱包来模拟用户（Joe，Alice，Bob和Gopesh）之间的交互。 在通过比特币网络跟踪交易到区块链时，我们将使用一个区块链接浏览器站点来显示每个步骤。 blockchain explorer是一个作为比特币搜索引擎运行的Web应用程序，它允许您搜索地址，交易和块，并查看它们之间的关系和资金流动。

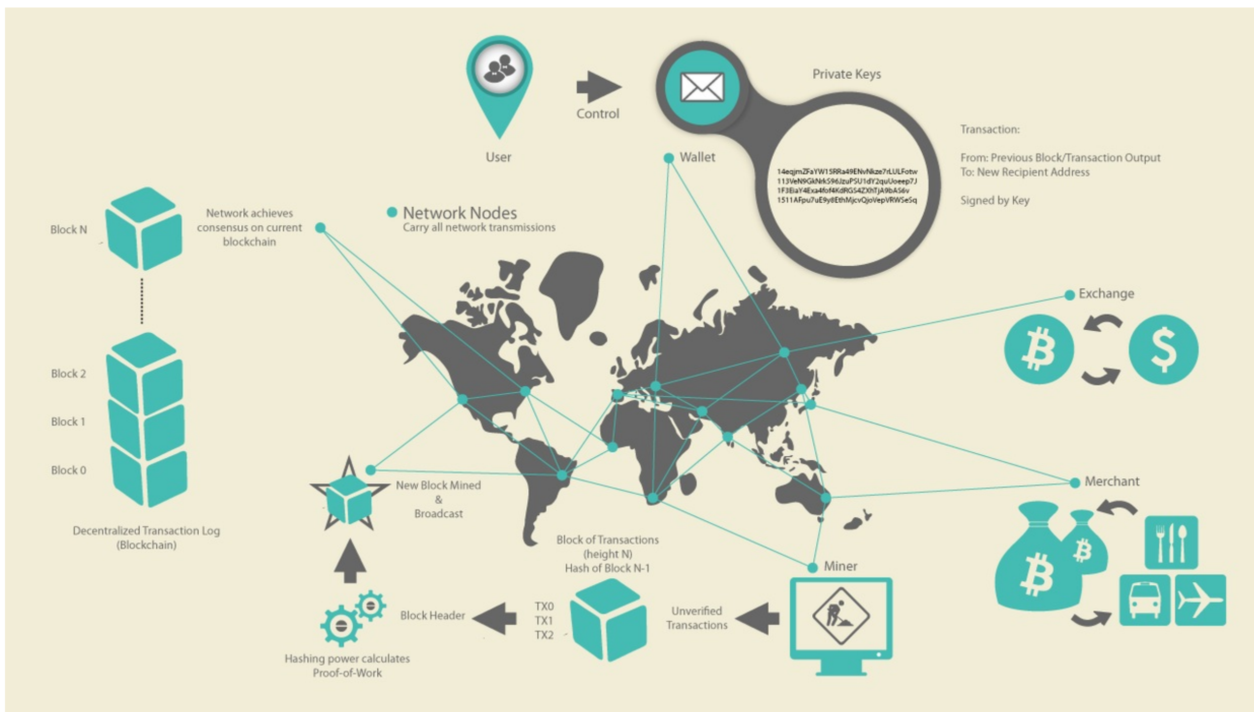


图2-1比特币网络概览

常见的区块链数据查询网站包括：

▷Bitcoin Block Explorer

▷BlockCypher Explorer▷blockchain.info

▷BitPay Insight

以上每一个查询网站都有搜索功能，可以通过地址，交易哈希值或区块号，搜索到在比特币网络和区块链中对应的数据。我们将给每个交易和区块例子提供一个链接，方便你做详细研究。

2.1.2 买咖啡

在之前章节里，Alice是一名刚刚获得第一枚比特币的新用户。在"1.4.2 获取你的第一枚比特币"一节中，Alice和她的朋 友Joe会面时，用现金换取了比特币。由Joe产生的这笔交易使得Alice的钱包拥有了0.10比特币。现在Alice将第一次使用比特币在加利福尼亚州帕罗奥图的Bob咖啡店买一杯咖啡。

Bob咖啡店给他的销售网点系统新增加了一个比特币支付选项，价格单上列的是当地货币（美元）的售价，但在收银台，顾客可以选择用美元或比特币支付。此时，Alice点了杯 咖啡，然后Bob将交易键入到收银机，之后销售系统将按照当前市场汇率把美元总价转换为比特币，然后同时显示两种货币的价格：

总价:

$1.50 USD

0.015 BTC

Bob说到，"总共1.50美元，或0.015 BTC比特币"

Bob的自助销售系统还将自动创建一个包含付款请求的二维码。

与一个简单包含目的比特币地址的二维码不同，当前支付请求的是一个二维编码过的URL，它包含有一个目的地址，一笔支付金额，和一个像"Bob咖啡"这样的交易描述。这使比特币钱包应用在发送支付请求时，可以预先填好支付用的特定信息，给用户显示一种友好易懂的描述。你可以用比特币钱包应用扫描这个二维码来看Alice可能看到的信息。



图2-2支付请求二维码

提示尝试用你的钱包扫描这个，看看地址和金额，但不要发送货币。

根据BIP0021的定义，这个付款二维码包括的URL的意思是：

```
bitcoin:1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA?
```

```
amount=0.015&
label=Bob%27s%20Cafe&
message=Purchase%20at%20Bob%27s%20Cafe
```

```
Components of the URL
```

```
A bitcoin address: "1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA"
The payment amount: "0.015"
A label for the recipient address: "Bob's Cafe"
A description for the payment: "Purchase at Bob's Cafe"
```

Alice用她的智能手机扫描了显示的条形码。她的智能手机显示一笔给Bob咖啡店的0.0150比特币的支付请求，然后她按下发送键授权了这笔支付。在几秒钟时间内（大约与信用卡授权所需时间相同）Bob将会在收银台看到这笔交易，并完成交易。在接下来的章节中，我们将更详细地检视这笔交易，观察Alice的钱包是怎样构建交易，交易又是怎样在网络中广播、怎样被验证，以及Bob在后续交易中怎样消费那笔钱。

注意从千分之一比特币(1毫比特币）到一亿分之一比特币（1聪比特币），比特币网络可以处理任意小额交易。在本书 中，我们将用“比特币”这个术语来表示任意数量的比特币货币，从最小单元（1聪）到可被挖出的所有比特币总数（21,000,000）。你可以像例1那样使用区块资源管理器站点来检查Alice与Bob's Cafe的交易:

例1. 查看Alice的交易

点击查看Alice的交易

2.2 比特币交易

简单来说，交易告知全网：比特币的持有者已授权把比特币转帐给其他人。而新持有者能够再次授权，转移给该比特币所有权链中的其他人，产生另一笔交易来花掉这些比特币，后面的持有者在花费比特币也是用类似的方式。

2.2.1交易输入输出

交易就像复式记账法账簿中的行。简单来说，每一笔交易包含一个或多个“输入”，输入是针对一个比特币账号的负债。 这笔交易的另一面，有一个或多个“输出”，被当成信用积分记入到比特币账户中。这些输入和输出的总额（负债和信用）不需要相等。相反，当输出累加略少于输入量时，两者的差额就代表了一笔隐含的“矿工费”，这也是将交易放进账簿的矿工所收集到的一笔小额支付。如图2-3描述的是一笔作为记账簿记录的比特币交易。交易也包含了每一笔被转移的比特币（输入）的所有权证明，它以所有者的数字签名形式存在，并可以被任何人独立验证。在比特币术语中，“消费”指的是签署一笔交易：转移一笔以前交易的比特币给以比特币地址所标识的新所有者。

Transaction as Double-Entry Bookkeeping

Inputs	Value	Outputs	Value
Input 1	0.10 BTC	Output 1	0.10 BTC
Input 2	0.20 BTC	Output 2	0.20 BTC
Input 3	0.10 BTC	Output 3	0.20 BTC
Input 4	0.15 BTC		
Total Inputs:	0.55 BTC	Total Outputs:	0.50 BTC

	Inputs	*0.55 BTC*
-	*Outputs*	*0.50 BTC*
	Difference	*0.05 BTC (implied transaction fee)*

图2-3交易就像复式记账

2.2.2交易链

Alice支付Bob咖啡时使用一笔之前的交易作为输入。在以前的章节中，Alice从她朋友Joe那里用现金换了点比特币。那笔交易创建了一些被Alice的密钥锁定的比特币。在她支付Bob咖啡店的新交易中使用了之前的交易作为输入，并以支付咖啡和找零作为新的输出。交易形成了一条链，最近交易的输入对应以前交易的输出。Alice的密钥提供了解锁之前交易输出的签

名，因此向比特币网络证明她拥有这笔钱。她将买咖啡的这笔支付附加到Bob的地址上，实现“阻止”那笔输出，明确指明要求是Bob签名才能消费这笔钱。这就描述了在Alice和Bob之间钱的转移。下图展示了从Joe到Alice再到Bob的交易链。

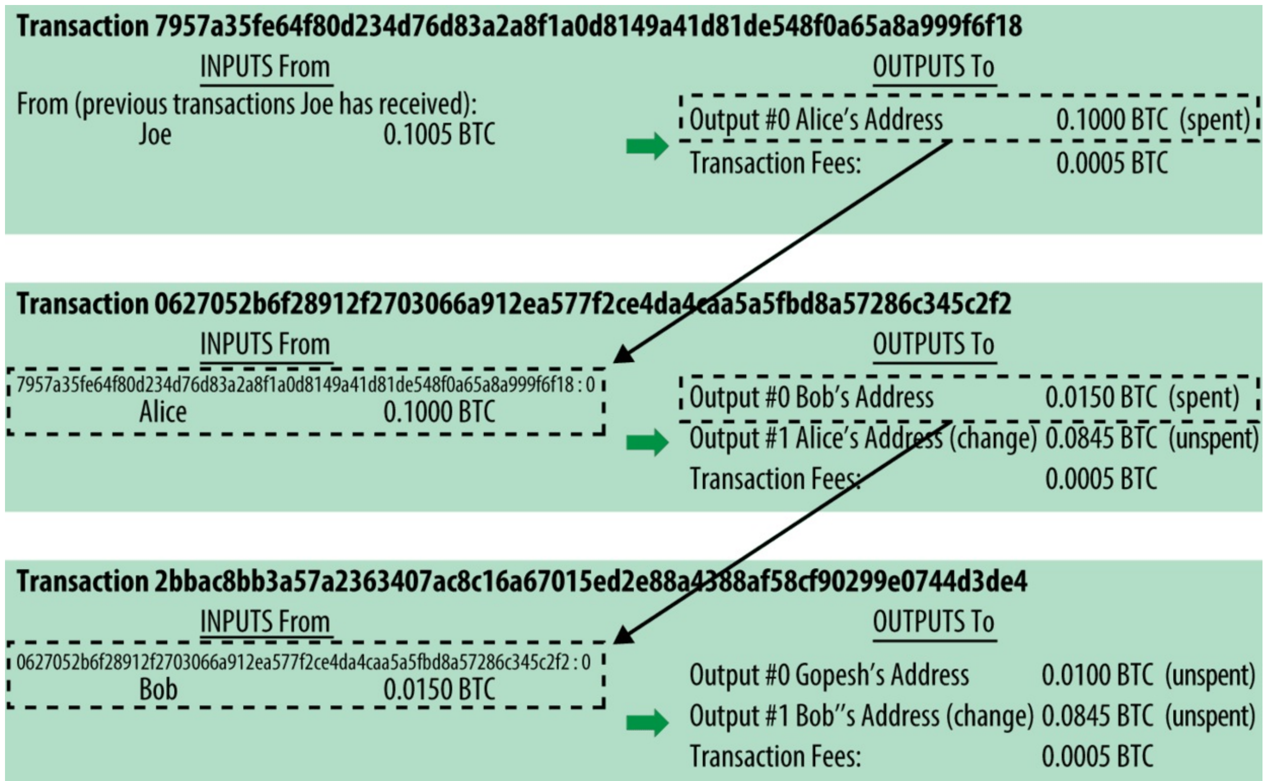


图2-5交易链中，一笔交易输出就是另一笔交易的输入

2.2.3找零

许多比特币交易都会包括新所有者的地址（买方地址）和当前所有者的地址（称为找零地址）的输出。这是因为交易输入，就像纸币那样能够，不能再分割。如果您在商店购买了5美元的商品，但是使用20美元的美金来支付商品，您预计会收到15美元的找零。相同的概念适用于比特币交易输入。如果您购买了一个价格为5比特币但只能使用20比特币输入的商品，那么您可以将5个比特币的一个输出发送给商店所有者，并将一个15比特币的输出返回给您自己作为找零（减去任何适用的交易费用）。重要的是，找零地址不必与输入时提供的地址相同，出于隐私的原因，通常是所有者钱包中的新地址。

不同的钱包可以在汇总输入以进行用户请求的付款时使用不同的策略。它们可能会聚合许多小输入，或者使用等于或大于所需付款的输入。除非钱包可以以这样的方式汇总输入，以便将所需付款与交易费用完全匹配，否则钱包将需要产生一些找零。这与人们如何处理现金非常相似。如果你总是在你的钱包支付时使用最大的面额，你会最终得到一个充满零钱的钱包。如果你只使用零钱，你最终会换来整钱。人们总是在潜意识中在这两个极端之间找到平衡，而比特币钱包开发商也力图实现这种平衡。总的来讲，交易是将钱从交易输入移至输出。输入是指钱币的来源，通常是之前一笔交易的输出。交易输出将约定金额发送到新的所

有者的比特币地址，并将找零输出返回原来原来所有者。 一笔交易的输出可以被当做另一笔新交易的输入，这样随着钱从一个地址被移动到另一个地址的同时形成了一条所有权链（如图2-4）。

2.2.4 常见的交易形式

最常见的交易形式是从一个地址到另一个地址的简单支付，这种交易也常常包含给支付者的“找零”。一般交易有一个输入和两个输出，如图2-5所示:

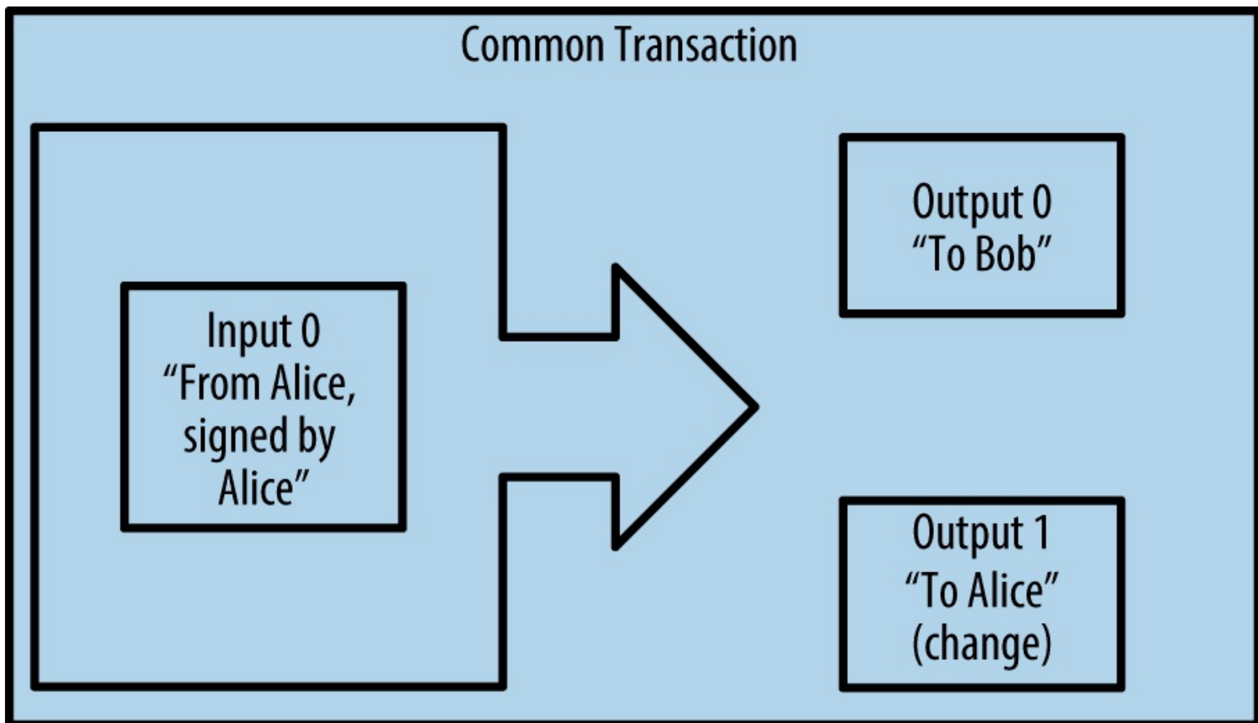


图2-5最普通的交易

另一种常见的交易形式是集合多个输入到一个输出（如图2-6）的模式。这相当于现实生活中将很多硬币和纸币零钱兑换为一个大额面钞。像这样的交易有时由钱包应用产生来清理许多在支付过程收到的小数额的找零。

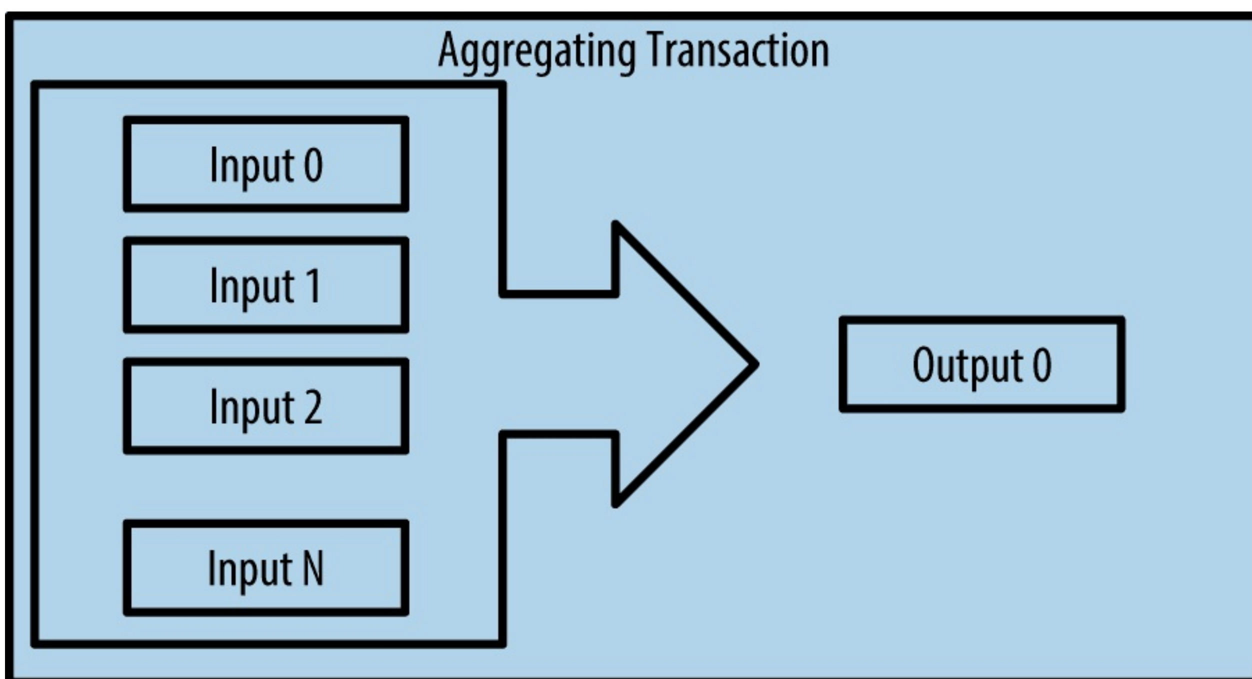


图2-6打包资金的交易

最后，另一种在比特币账簿中常见的交易形式是将一个输入分配给多个输出，即多个接收者（如图2-7）的交易。这类交易有时被商业实体用作分配资金，例如给多个雇员发工资的情形。

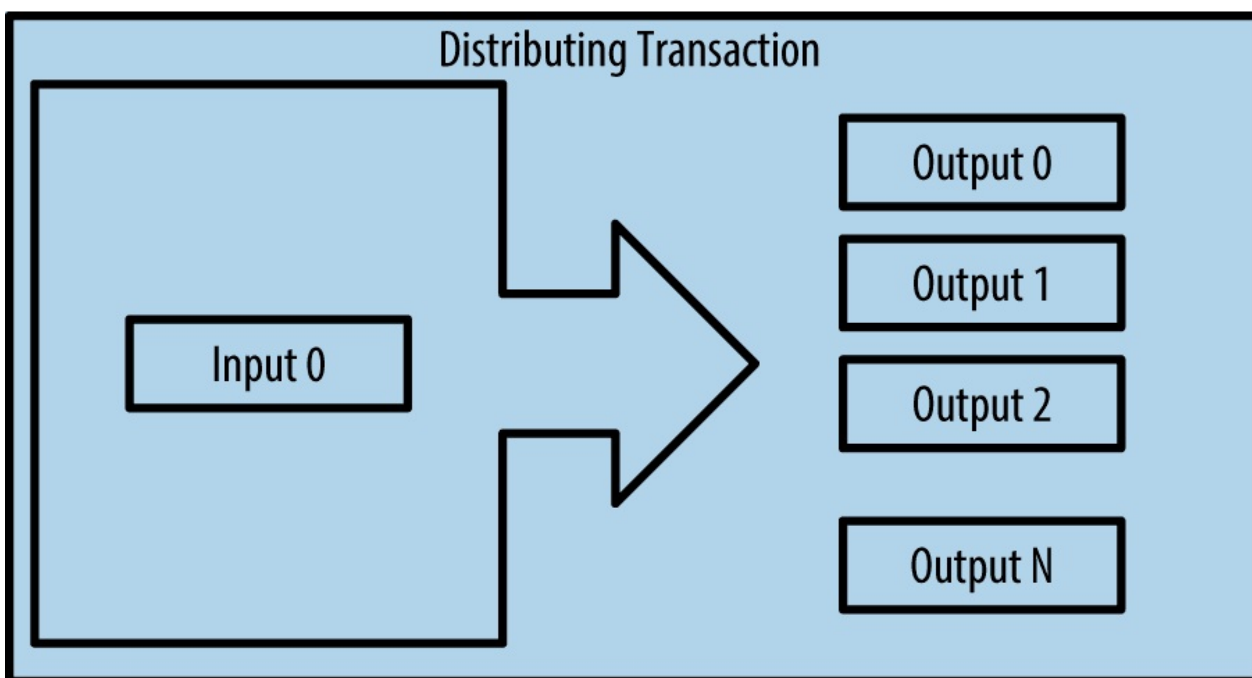


图2-7分散资金的交易

2.3 交易的构建

Alice的钱包应用知道如何选取合适的输入和输出以建立Alice所希望的交易。Alice只需要指定目标地址和金额，其余的细节钱包应用会在后台自动完成。很重要的一点是，钱包应用甚至可以在完全离线时建立交易。就像在家里写张支票， 之后放到信封发给银行一样，比特币交易建立和签名时不用连接比特币网络。只有在执行交易时才需要将交易发送到网络。

2.3.1 获取正确的输入

Alice的钱包应用首先要找到一些足够支付给Bob所需金额的输入。大多数钱包应用跟踪着钱包中某个地址的所有可用输出。因此Alice的钱包会包含她用现金从Joe那里购买的比特币的交易输出副本（参见 在“获取你的第一枚比特币 ”一节）。完整客户端含有整个区块链中所有交易的所有未消费输出副本。这使得钱包既能拿这些输出构建交易，又能在收到新交易时很快地验证其输入是否正确。然而，完整客户端占太大的硬盘空间，所以大多数钱包使用轻量级的客户端，只保存用户自己的未消费输出。

如果钱包客户端没有某一未消费交易输出，它可以通过不同的服务者提供的各种API或完整索引节点的JSON PRC API从比特币网络中拿到这一交易信息。例子2-1展示了用HTTP GET命令对一个特定URL建立了一个API的请求。这 个URL会返回一个地址的所有未消费交易输出，以提供给需要这些信息的任何应用作为建立新交易的输入而进行消费。 我们用一个简单的HTTP命令行客户端 cURL来获得这个响应数据。

例2-1 查找Alice的比特币地址所有的未消费的输出

```
$ curl https://blockchain.info/unspent?active=1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK
{

    "unspent_outputs":[

        {
            "tx_hash":"186f9f998a5...2836dd734d2804fe65fa35779",
            "tx_index":104810202,
            "tx_output_n": 0,
            "script":"76a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac",
            "value": 10000000,
            "value_hex": "00989680",
            "confirmations":0
        }

    ]
}
```

例2-2的响应数据显示了在Alice的地址 1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK 上面有一个未消费输出（还未被兑换）。这个响应包含一个交易的引用。而从Joe那里转过来的未消费输入就包含在这个交易里面，它的价值是一千万聪 （satoshi），即 0.10比特币。通过这个信息，Alice的钱包应用就可以创建新的交易将钱转账到新地址。

提示点击这里 查看Joe和Alice间的交易信息。

如你所见，Alice的钱包在单个未消费的输出中有足够的比特币支付一杯咖啡。假如不够的话，Alice的钱包应用就不得不搜寻一些小的未消费输出，像是从一个存钱罐里找硬币一样，直到找到足够支付咖啡的数量。在两种情境下，可能都需要找回零钱，而这些找零也会是钱包所创建的交易的输出组成部分。我们会在下一节会有所描述。

2.3.2 创建交易输出

交易的输出会被创建成为一个包含这笔数额的脚本的形式，只能被引入这个脚本的一个解答后才能兑换。简单点说就 是，Alice的交易输出会包含一个脚本，这个脚本说“这个输出谁能拿出一个签名和Bob的公开地址匹配上，就支付给谁”。因为只有Bob的钱包的私钥可以匹配这个地址，所以只有Bob的钱包可以提供这个签名以兑换这笔输出。因此Alice 会需要Bob的签名来包装一个输出。

这个交易还会包含第二个输出。因为Alice的金额是0.10比特币的输出形式，对0.015 比特币一杯的咖啡来说太多了，需要找Alice 0.085比特币的零钱。Alice钱包创建给她的零钱的支付就在付给Bob的同一个交易里面。可以说，Alice的钱包将她的金额分成了两个支付：一个给Bob，一个给自己。她可以在以后的交易里消费这笔零钱输出。

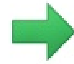
最后，为了让这笔交易尽快地被网络处理，Alice的钱包会多付一小笔费用。这个不是明显地包含在交易中的；而是通过输入和输出的差值所隐含的。如果Alice创建找零时只找 0.0845比特币，而不是 0.085比特币的话，这里就有剩下 0.0005比特币（50万聪） 。因为加起来小于0.10，所以这个 0.10 比特币的输入就没有被完整的消费了。这个差值会就被矿工当作交易费放到区块的交易里，最终放进区块链帐薄中。

这个交易的结果信息可以用区块链数据查询站点看到，如图2-8所示。

Transaction View information about a bitcoin transaction

0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2

1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK (0.1 BTC - Output)



1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA - (Unspent) 0.015 BTC
1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK - (Unspent) 0.0845 BTC

97 Confirmations 0.0995 BTC

Summary	
Size	258 (bytes)
Received Time	2013-12-27 23:03:05
Included In Blocks	277316 (2013-12-27 23:11:54 +9 minutes)

Inputs and Outputs	
Total Input	0.1 BTC
Total Output	0.0995 BTC
Fees	0.0005 BTC
Estimated BTC Transacted	0.015 BTC

图2-8 Alice和Bob咖啡店的交易

提示点击这里查看Alice支付Bob咖啡的交易的信息

2.3.3 将交易放到总账簿中

这个被Alice钱包应用创建的交易大小为258字节，包含了确认资金所有权和分配给新所有者所需要的全部信息。现在，这个交易必须要被传送到比特币网络中以成为分布式账簿（区块链）的一部分。在下一节里，我们来看下一个交易如何成为新区块的一部分， 以及区块是如何被挖矿构建的。最后，我们会看看新区块被加进区块链后，是如何随更多区块的添加而增加可信度的。

2.3.3.1 交易的传送

因为这个交易包含处理所需的所有信息，所以这个交易是被如何或从哪里传送到比特币网络的就无所谓了。比特币网络是由参与的比特币客户端联接几个其他比特币客户端组成的P2P网络。比特币网络的目的是将交易和区块传播给所有参与者。

2.3.3.2 如何传播

“说着”比特币协议，从而实现参与比特币网络的任何系统（例如服务器，桌面应用程序或钱包）都称为比特币节点。Alice的钱包应用可以发送新的交易给其它任意一个已联接到互联网的比特币客户端，不论其是由有线网、WiFi、还是通过手机联接的。她的钱包不必直接连着

Bob的比特币钱包，且她不必使用咖啡厅提供的网络联网，虽然这两者都是可能的。任何比特币网络节点（其它客户端）收到一个之前没见过的有效交易时会立刻将它转发给联接到自身的其它节点。 因此，这个交易迅速地从P2P网络中传播开来，几秒内就能到达大多数节点。

2.3.3.3 Bob的视角

如果Bob的比特币钱包应用是直接连接Alice的钱包应用的话，Bob的钱包应用也许就是第一个收到这个交易的节点。然 而，即使Alice的交易是从通过其它节点发过来的，一样可以在几秒钟内到达Bob钱包应用这里。Bob的钱包会立即确认 Alice的交易是一个收入支付，因为它包含能用Bob的私钥兑换的输出。Bob的钱包应用也能够独立地用之前未消费输入来确认这个交易是正确构建的，并且由于包含足够交易费会被下一个区块包含进去。这时Bob只需冒很小的风险，因为这个交易会很快被加到区块且被确认。

提示一个对比特币交易的常见误解是它们必须要等10分钟后被确认加进一个新区块，或等60分钟以得到六次确认后才是有效的。虽然这些确认可以确保交易已被整个网络接受，但对于像一杯咖啡这样的小额商品来说就没有必要等待那么长时间了。一个商家可以免确认来接受比特币小额支付。这样做的风险不比接受一个不是用有效身份证领取或没有签名的信用卡的风险更大，而后者是现在商家常做的事情。

2.4 比特币挖矿

这个交易现在在比特币网络上传播开来。但只有被一个称为挖矿的过程验证且加到一个区块中之后，这个交易才会成为 这个共享账簿（区块链）的一部分。关于挖矿的详细描述请见第10章。比特币系统的信任是建立在计算的基础上的。交易被包在一起放进区块中时需要极大的计算量来证明，但只需少量计算就能验证它们已被证明。

挖矿在比特币系统中有两个重要作用：

▷ 挖矿节点通过参考比特币的共识规则验证所有交易。 因此，挖矿通过拒绝无效或畸形交易来提供比特币交易的安全性。

▷ 挖矿在构建区块时会创造新的比特币，和一个中央银行印发新的纸币很类似。每个区块创造的比特币数量是固定的， 随时间会渐渐减少。

挖矿在成本和报酬之间取得了良好的平衡。 挖矿采用电力来解决数学问题。 一个成功的矿工将以新的比特币和交易费的形式获取奖励。 但是，只有矿工正确验证了所有的交易，才能获得奖励，才能达到协商一致的规则。 这种微妙的平衡为没有中央权力机构的比特币提供安全保障。

描述挖矿的一个好方法是将之类比为一个巨大的多人数独谜题游戏。一旦有人发现正解之后，这个数独游戏会自动调整难度以使游戏每次需要大约10分钟解决。想象一个有几千行几千列的巨大数独游戏。如果给你一个已经完成的数独， 你可以很快地验证它。然而，如果这

个数独只有几个方格里有数字其余方格都为空的话，就会花费非常长的时间来解决。这个数独游戏的困难度可以通过改变其大小（更多或更少行列）来调整，但即使它非常大时验证它也是相当容易的。而比特币中的 "谜题" 是基于哈希加密算法的，其展现了相似的特性：非对称地，它解起来困难而验证很容易，并且它的困难度可以调整。

在 “比特币的应用、用户和他们的故事”一节中，我们提到了一个叫Jing的在上海学计算机工程的学生。Jing在比特币网络中扮演了一个矿工的角色。大概每10分钟，Jing和其他上千个矿工一起展开一场对一个区块的交易寻找正解的全球竞赛。为寻找这个解，也被称为工作量证明，整个网络需要具有每秒亿万次哈希计算的能力。这个工作量证明算法指的用SHA256加密算法不断地对区块头和一个随机数字进行哈希计算，直到出现一个和预设值相匹配的解。第一个找到这个解的矿工会赢得这局竞赛并会将此区块发布到区块链中。

Jing从2010年开始挖矿，当时他使用一个非常快的桌面电脑来为新区块寻找正解。随着更多的矿工加入比特币网络中， 寻找谜题正解的困难度迅速增大。不久，Jing和其他矿工升级成更专业的硬件，比如游戏桌面电脑或控制台专用的高端 独享图像处理单元芯片（即显卡GPU）。在写这本书的时候，解题已经变得极其困难，只有使用集成了几百个挖矿专用算法硬件并能同时在一个单独芯片上并行工作的专用集成电路（ASIC）挖矿才会营利。Jing的公司同时加入了一个类似彩票奖池的、能够让多个矿工共享计算力和报酬的矿池。Jing现在运行两个通过USB联接的ASIC机器每天24小时不间断地挖 矿。他卖掉一些挖矿所得到的比特币来支付电费，通过收益获得一些收入。

2.5 区块中的挖矿交易记录

新交易不断地从用户钱包和应用流入比特币网络。当比特币网络上的节点看到这些交易时，会先将它们放到各自节点维护的一个临时的未经验证的交易池中。当矿工构建一个新区块时， 会将这些交易从这个交易池中拿出来放到这个新区块中，然后通过尝试解决一个非常困难的问题（也叫工作量证明）以证明这个新区块的合法性。挖矿过程的细节会在“挖矿简介”一节中详加描述。

这些交易被加进新区块时，以交易费用高的优先以及其它的一些规则进行排序。矿工一旦从网络上收到一个新区块时， 会意识到在这个区块上的解题竞赛已经输掉了，会马上开始下一个新区块的挖掘工作。它会立刻将一些交易和这个新区块的数字指纹放在一起开始构建下一个新区块，并开始给它计算工作量证明。每个矿工会在他的区块中包含一个特殊的交易，将新生成的比特币（当前每区块为12.5比特币）作为报酬支付到他自己的比特币地址，再加上块中所有交易的交易费用的总和作为自己的报酬。如果他找到了使得新区块有效的解法，他就会得到这笔报酬，因为这个新区块被加入到了总区块链中，他添加的这笔报酬交易也会变成可消费的。 参与矿池的Jing设置了他的软件，使得构建新区块时会将报酬地址设为矿池的地址。然后根据各自上一轮贡献的工作量将所得的报酬分给Jing和其他参与矿池挖矿的矿工。

Alice的交易被网络拿到后放进未验证交易池中。一旦被挖矿软件验证，它就被包含在由Jing的采矿池生成的新块（称为候选块）中。 参与该采矿池的所有矿工立即开始计算候选块的工作证明。大约 在Alice的钱包第一次将这个交易发送出来五分钟后，Jing的ASIC矿机发现了新区块的正解并将这个新区块发布到网络上后，一旦被其它矿机验证，它们就会立即投身到构建新区块的竞赛中。

Jing的ASIC矿机发现了新区块的正解并将之发布为第277,316号区 块，包含420个交易，包括Alice的交易。包含Alice交易的区块对这个交易来说算一次"确认"。

提示你可以查看包含Alice交易记录的这个区块的信息。

大约19分钟后，第277,317号新区块诞生在另一个挖矿节点中。因为这个新区块是在包含Alice交易的第277,316号区块的上层（栈），在这个区块的基础上增加了更多的计算，因此就加强了这些交易的可信度。基于这个区块每产生一个新区块，对这个交易来说就会增加了一次"确认"。当区块一个个堆上来时，这个交易变得指数级地越来越难被推翻，因此它在网络中得到更多信任。

在图2-9中，我们可以看到包含Alice的交易的第277,316号区块。在它之下有377,361个区块（包括0号区块），像链子 一样一个连着一个（区块链），一直连到0号区块，即创世区块。随着时间变长，这个区块链的高度也随之增长，每个区块和整个链的计算复杂度也随之增加。包含Alice的交易的区块后面形成的新区块使得信任度进一步增加，因为他们叠加了更多的计算在这个越来越长的链子上。按惯例来说，一个区块获得六次以上“确认”时就被认为是不可撤销的了，因为要撤销和重建六个区块需要巨量的计算。在第10章我们会详细描述挖矿和信任建立的过程。

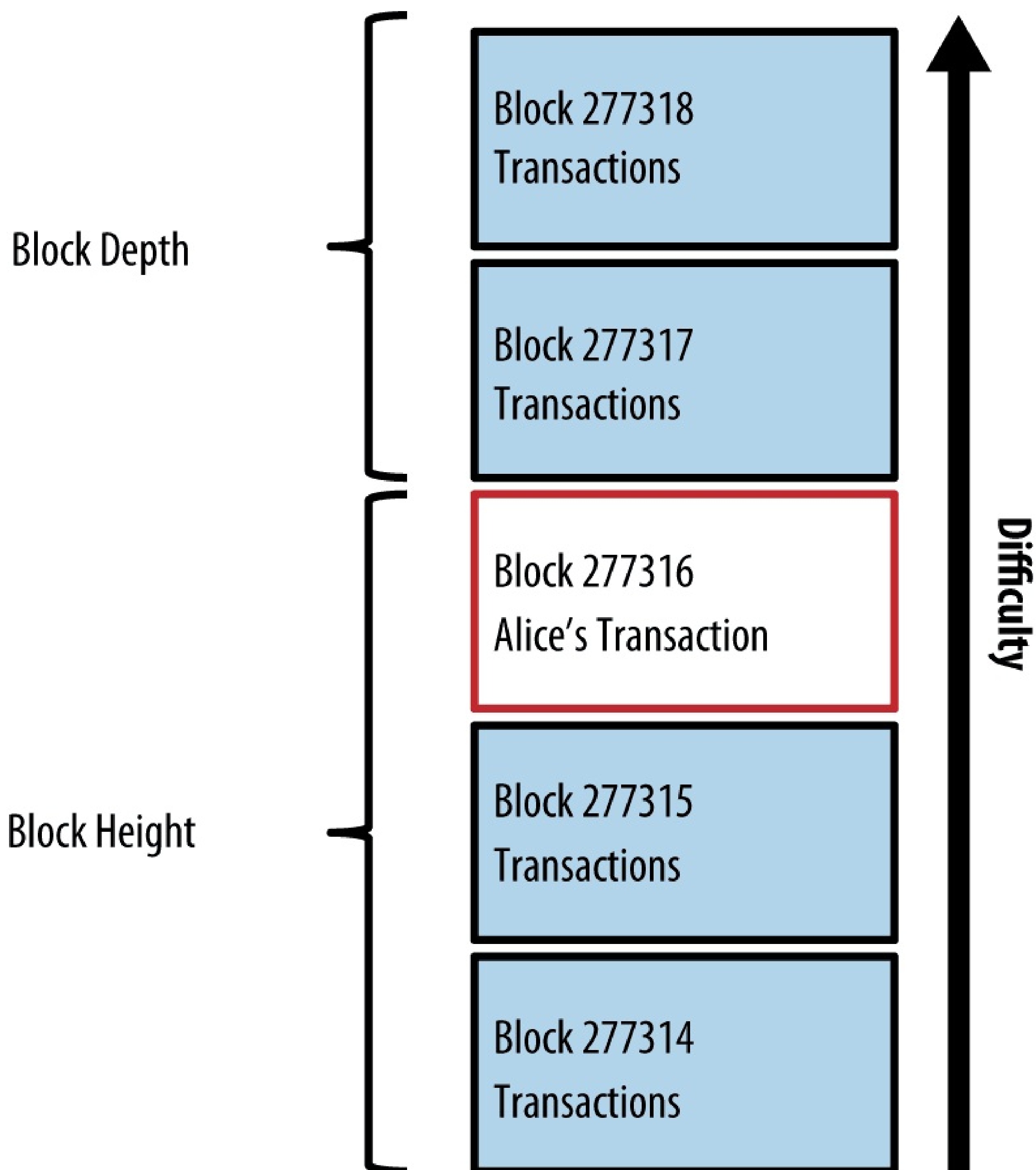


图2-9Alice的交易包括在区块277316

2.6 消费这笔交易

既然Alice的这笔交易已经成为区块的一部分被嵌入到了区块链中，它就成为了整个分布式比特币账簿的一部分并对所有比特币客户端应用可见。每个比特币客户端都能独立地验证这笔交易是有效且可消费的。全节点客户端可以追溯钱款的来源，从第一次有比特币在区块里生

成的那一刻开始，按交易与交易间的关系顺藤摸瓜，直到Bob的交易地址。轻量级客户端通过确认一个交易在区块链中且在它后面有几个新区块来确认一个支付的合法性。这种方式叫做简易支付验证（参 见“简易支付验证（SPV）节点”）。

Bob现在可以将此交易和其它交易的结果信息作为输入，创建新的所有权为其他人的交易。这样就实现了对此交易的消费。举个例子，Bob可以用Alice支付咖啡的比特币转账给承包商或供应商以支付相应费用。大多数情况下，Bob用的比特币客户端会将多个小额支付聚合成一个大的支付，也许会将一整天的比特币收入聚合成一个交易。这样会将多个支付合成到咖啡店财务账户的一个单独地址。图2-10为交易集合示例。

当Bob花费从Alice和其他顾客那里赚得的比特币时，他就扩展了比特币的交易链条。而这个链条会被加到整个区块链账簿，使所有人知晓并信任。我们假定Bob向在邦加罗尔的网站设计师Gopesh支付一个新网页的设计费用。那么区块交易链如图2-10所示。

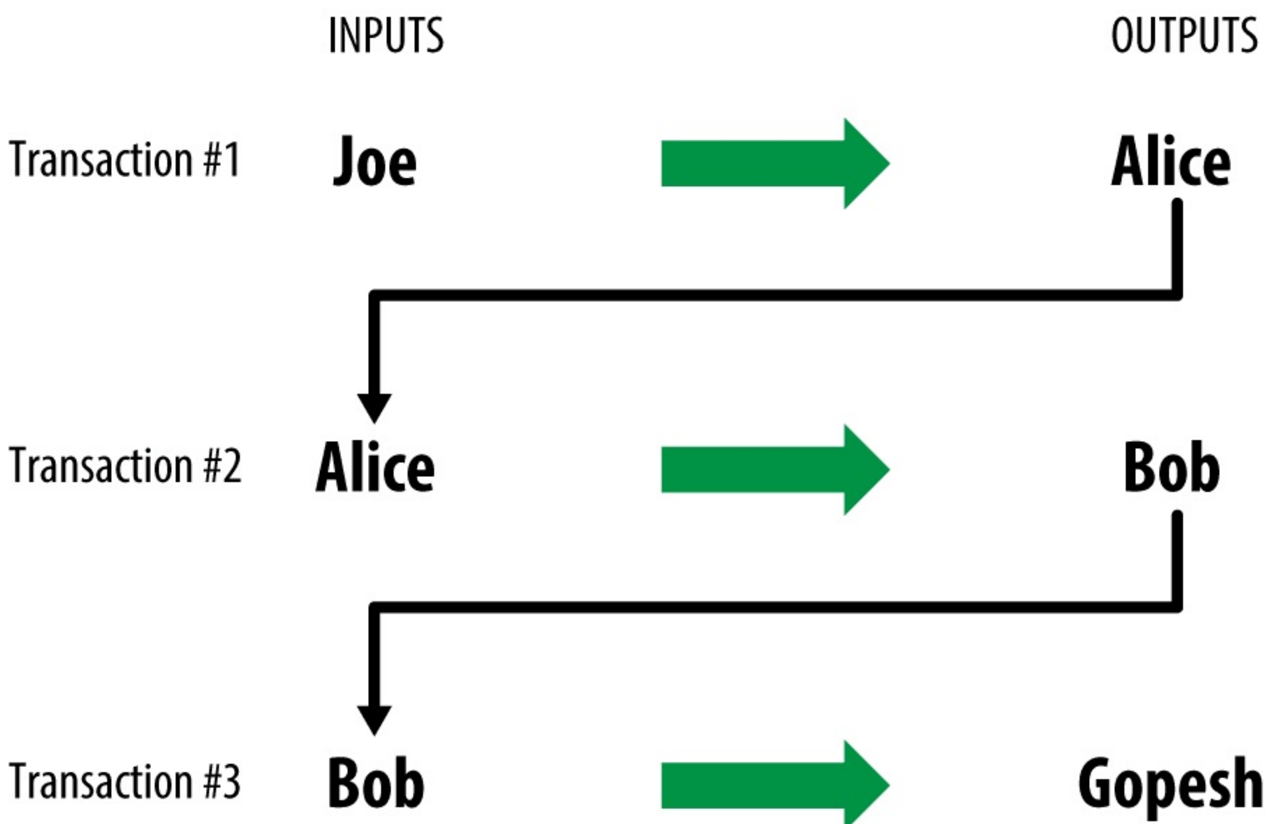


图2-10Alice的交易成为 Joe 和 Gopesh交易的一部分

在本章中，我们看到了交易如何被构建为一个链，并将价值从一个所有者转移到所有者。 我们还追踪了Alice的交易，从她的钱包中创建的那一刻起，通过比特币网络被矿工记录在区块链。 在本书的其余部分，我们将研究钱包，地址，签名，交易，网络和最终挖矿等背后的具体技术。

Bitcoin是一个开源项目，源代码可以根据开放（MIT）许可证提供，可免费下载并用于任何目的。 开源意味着不仅仅是自由使用。 这也意味着比特币是由一个开放的志愿者社区开发的。 起初这个社区只有中本聪。 到2016年，比特币的源代码有超过400个贡献者，大约十几位开发人员几乎全职工作，几十名开发人员兼职。 任何人都可以为代码贡献 - 包括你！

当由中本聪创建比特币时，软件实际上是在后来大名鼎鼎的[satoshi_whitepaper]白皮书之前完成的。 中本聪想在写作之前确保它有效工作。 那么这个第一个实践，就叫做“比特币（Bitcoin）”或者“Satoshi客户”，实际上已经被大大的修改和改进了。它已经演变成所谓的比特币核心，以区别于其他兼容的实现方式。 比特币核心是比特币系统的参考实现，这意味着它是如何实施的权威参考。 Bitcoin Core实现了比特币的所有方面，包括钱包，交易和区块验证引擎，以及P2P网络中的完整网络节点。

警示 即使Bitcoin Core包含钱包的参考实现，但这并不意味着可以用作用户或应用程序的生产钱包。 建议应用程序开发人员使用现代标准（如BIP-39和BIP-32）构建钱包（请参阅助记词]和[hd钱包]章节）。 BIP就是比特币改进提案（Bitcoin Improvement Proposal）。

下图为比特币核心的架构。

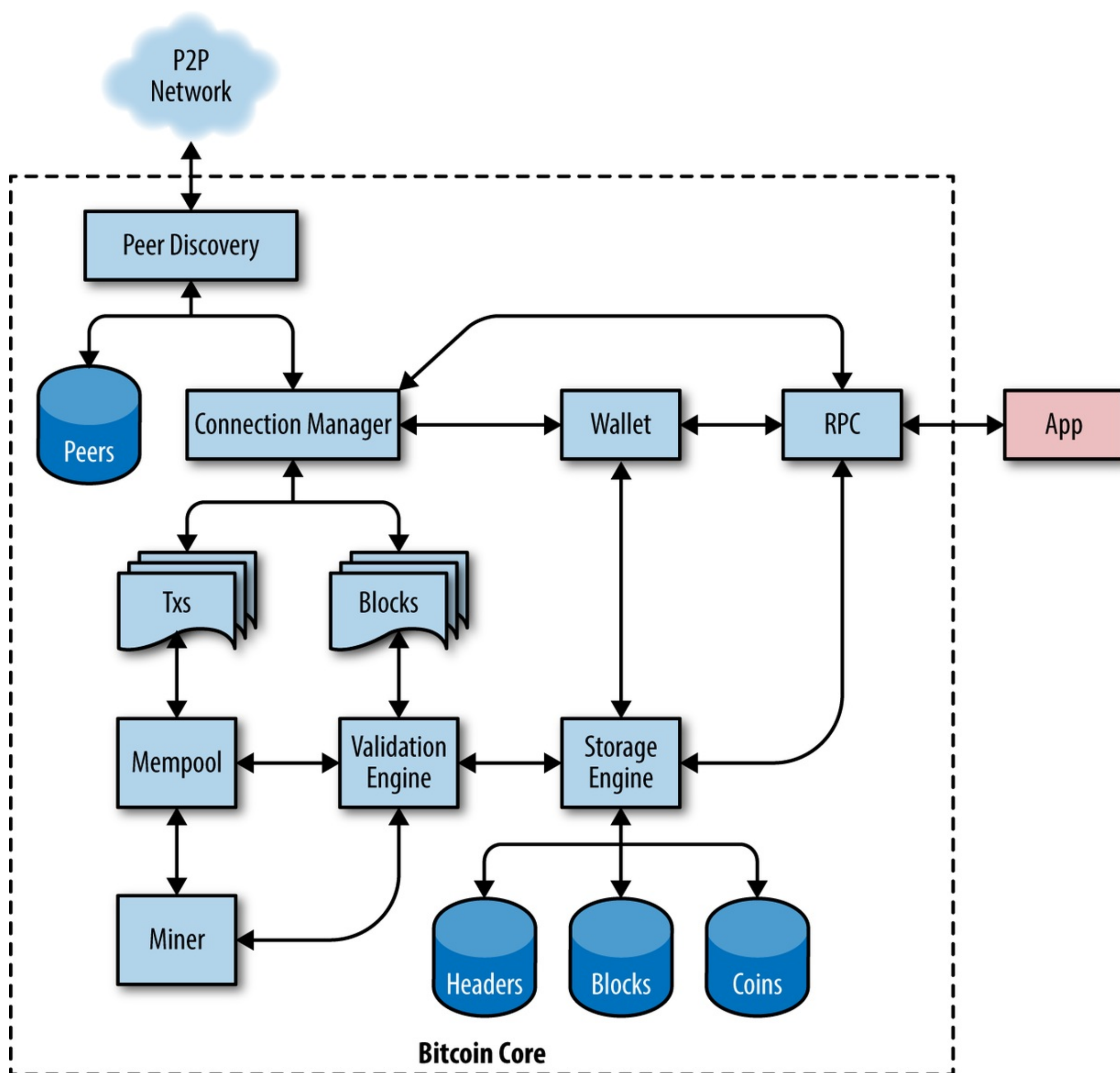


图3-1比特币核心架构（来源Eric Lombrozo）

3.1比特币开发环境

如果您是开发人员，您将需要使用所有工具，库和支持软件来设置开发环境，以编写比特币应用程序。 这一章涉及的技术细节较深，我们将逐步介绍该过程。 如果你觉得过于繁琐（并且您实际上并没有设置开发环境），建议你跳到下一章，技术性比本章浅显一些。

3.2从源码编译比特币核心

Bitcoin Core的源代码可以作为ZIP存档下载，也可以从GitHub克隆权威的源代码库。 在GitHub比特币页面GitHub bitcoin page上，选择"下载ZIP"。 或者，使用git命令行在系统上创建源代码的本地副本。

提示在本章的许多例子中，我们将使用操作系统的命令行界面（也称为“shell”），通过“terminal”应用程序访问。 shell将显示提示你键入命令，并且shell反馈一些文本和一个新的提示您的下一个命令。 提示符可能在您的系统上看起来不同，但在以下示例中，它由符号表示（非用户）。在示例中，当您在符号后面看到文本时，不要键入符号，而是在其后面键入命令，然后按键执行该命令。在示例中，每个命令下面的行是操作系统对该命令的响应。当你看到下一个前缀时，应该继续输入下一个新的命令，可以一直重复这个过程。

在本例中，我们使用git命令来创建源代码的本地副本（“clone”）：

```
$ git clone https://github.com/bitcoin/bitcoin.git
Cloning into 'bitcoin'...
remote: Counting objects: 66193, done.
remote: Total 66193 (delta 0), reused 0 (delta 0), pack-reused 66193
Receiving objects: 100% (66193/66193), 63.39 MiB | 574.00 KiB/s, done.
Resolving deltas: 100% (48395/48395), done.
Checking connectivity... done.
$
```

提示Git是最广泛使用的分布式版本控制系统，是任何软件开发人员工具包的重要组成部分。您可能需要在操作系统上安装git命令或git的图形用户界面。

当git克隆操作完成后，您将在目录比特币中拥有源代码存储库的完整本地副本。 在提示符下键入“cd bitcoin”，进入为此目录：

$ cd bitcoin

3.2.1选择比特币核心版本

默认情况下，本地副本将与最新的代码同步，这可能是不稳定的或Beta版的比特币。 在编译代码之前，先查看一个发布标签tag，选择一个特定的版本。 这将使本地副本与关键字标签所标识的代码库的特定快照同步。 开发人员使用标签来标记版本号的特定版本的代码。 首先，要找到可用的标签，我们使用git tag命令：

```
$ git tag
v0.1.5
v0.1.6test1
v0.10.0
...
v0.11.2
v0.11.2rc1
v0.12.0rc1
v0.12.0rc2
...
```

tag列表显示所有发布的比特币版本。 根据惯例，用于测试的发布候选版本具有后缀“rc”。 可以在生产系统上运行的稳定版本没有后缀。 从上面的列表中，选择最高版本的版本，在编写时是v0.11.2。 要使本地代码与此版本同步，请使用git checkout命令：

```
$ git checkout v0.11.2
HEAD is now at 7e27892... Merge pull request #6975
```

您可以通过输入命令git status来确认您有所需的版本“checkout”：

```
$ git status
HEAD detached at v0.11.2
nothing to commit, working directory clean
```

3.2.2配置构建比特币核心

源代码中包括文档，可以在多个文件中找到。通过在提示符下键入“more README.md”并使用空格键进入下一页，查看bitcoin目录中README.md中的主要文档。在本章中，我们将在Linux上构建命令行比特币客户端，也称为比特币（bitcoind）。在系统中查看编译bitcoind命令行客户端的说明，方法是输入“more doc / build-unix.md” 。可以在doc目录中找到macOS和Windows的替代说明，分别为build-osx.md或build-windows.md。

仔细查看build前提条件，这些前提是build文档的第一部分。这些是在您开始编译比特币之前必须存在于系统上的库。如果缺少这些先决条件，build过程将失败并显示错误。如果发生这种情况是因为您缺失先决条件，则可以安装它，然后从您所在的地方恢复build过程。假设安装了先决条件，您可以通过使用autogen.sh脚本生成一组build脚本来启动build过程。

注意Bitcoin Core build过程已经从0.9开始更改为使用autogen / configure / make系统。 旧版本使用简单的Makefile，与以下示例的方法略有不同。 因此要按照要编译的版本的说明进行操作。 在0.9中引入的autogen / configure / make可能是用于所有未来版本代码的build系统，并且是以下示例中演示的系统。

```
$ ./autogen.sh
...
glibtoolize: copying file 'build-aux/m4/libtool.m4'
glibtoolize: copying file 'build-aux/m4/ltoptions.m4'
glibtoolize: copying file 'build-aux/m4/ltsugar.m4'
glibtoolize: copying file 'build-aux/m4/ltversion.m4'
...
configure.ac:10: installing 'build-aux/compile'
configure.ac:5: installing 'build-aux/config.guess'
configure.ac:5: installing 'build-aux/config.sub'
configure.ac:9: installing 'build-aux/install-sh'
configure.ac:9: installing 'build-aux/missing'
Makefile.am: installing 'build-aux/depcomp'
...
```

autogen.sh脚本创建一组自动配置脚本，它会询问系统以发现正确的设置，并确保您拥有编译代码所需的所有库。 其中最重要的是配置脚本，它提供了许多不同的选项来自定义构建过程。 键入“./configure --help”查看各种选项：

```
$ ./configure --help
`configure' configures Bitcoin Core 0.11.2 to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...

...
Optional Features:
  --disable-option-checking  ignore unrecognized --enable/--with options
  --disable-FEATURE       do not include FEATURE (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]

  --enable-wallet         enable wallet (default is yes)

  --with-gui[=no|qt4|qt5|auto]
...
```

配置脚本允许您通过使用--enable-FEATURE和--disable-FEATURE标志来启用或禁用bitcoind的某些功能，其中FEATURE由功能名称替换，如帮助输出中所列。 在本章中，我们将构建具有所有默认功能的bitcoind客户端。 我们不会使用配置标志，但您应该查看它们以了解可选功能是客户端的一部分。 如果您处于学术环境中，计算机实验室的限制可能需要您在主目录中安装应用程序（例如，使用--prefix = $ HOME）。

以下是一些有用的选项，可以覆盖configure脚本的默认行为：

--prefix=$HOME

这将覆盖生成的可执行文件的默认安装位置（它是/ usr / local /）。 使用$ HOME将所有内容放在您的主目录或不同的路径中。

--disable-wallet

这用于禁用参考钱包的实现。

--with-incompatible-bdb

如果您正在构建钱包，请允许使用不兼容版本的Berkeley DB库。

--with-gui=no

不要构建图形用户界面，图形界面需要Qt库。 这只构建服务器和命令行。

接下来，运行configure脚本来自动发现所有必需的库，并为您的系统创建一个自定义的构建脚本：

```
$ ./configure
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
...
[many pages of configuration tests follow]
...
$
```

如果一切顺利，configure命令将会以创建可定制的构建脚本结束。这些构建脚本允许我们编译bitcoind。如果有缺失的库或是错误，configure命令将会以错误信息终止。如果出现了错误，可能是因为缺少库或是有不兼容的库。重新检查构建文档，确认你已经安装缺失的必备条件。然后运行configure，看看错误是否消失了。

3.2.3构建Bitcoin核心可执行文件

下一步，你将编译源代码，这个过程根据CPU和内存资源不同，但一般可能需要1个小时完成。在编译的过程中，你应该过几秒或是几分钟看一下输出结果。如果出现了问题，你会看到错误。如果中断了，编译的过程可以在任何时候恢复。输入make命令就可以开始编译了：

```
$ make
Making all in src
  CXX      crypto/libbitcoinconsensus_la-hmac_sha512.lo
  CXX      crypto/libbitcoinconsensus_la-ripemd160.lo
  CXX      crypto/libbitcoinconsensus_la-sha1.lo
  CXX      crypto/libbitcoinconsensus_la-sha256.lo
  CXX      crypto/libbitcoinconsensus_la-sha512.lo
  CXX      libbitcoinconsensus_la-hash.lo
  CXX      primitives/libbitcoinconsensus_la-transaction.lo
  CXX      libbitcoinconsensus_la-pubkey.lo
  CXX      script/libbitcoinconsensus_la-bitcoinconsensus.lo
  CXX      script/libbitcoinconsensus_la-interpreter.lo

[... many more compilation messages follow ...]

$
```

如果一切顺利，bitcoind现在已经编译完成。最后一步就是通过sudo make install 命令，安装bitcoind 可执行文件到你的系统路径下，可能会提示您输入用户密码，因为此步骤需要管理员权限：

```
$ sudo make install
Password:
Making install in src
 ../build-aux/install-sh -c -d '/usr/local/lib'
libtool: install: /usr/bin/install -c bitcoind /usr/local/bin/bitcoind
libtool: install: /usr/bin/install -c bitcoin-cli /usr/local/bin/bitcoin-cli
libtool: install: /usr/bin/install -c bitcoin-tx /usr/local/bin/bitcoin-tx
...
$
```

bitcoind 默认的安装位置是/usr/local/bin。你可以通过询问系统下面2个可执行文件的路径，来确认bitcoin是否安装成功。

```
$ which bitcoind
/usr/local/bin/bitcoind

$ which bitcoin-cli
/usr/local/bin/bitcoin-cli
```

3.2.4运行比特币核心节点

比特币的对等网络由网络“节点”组成，主要由志愿者和一些构建比特币应用程序的商业机构运行。 那些运行的比特币节点具有直接和权威的比特币区块链视图，并且具有所有交易的本地副本，由其自己的系统独立验证。 通过运行节点，您不必依赖任何第三方来验证交易。 此

外，通过运行比特币节点，您可以通过使其更健壮的方式为比特币网络做出贡献。

但是，运行节点需要一个具有足够资源来处理所有比特币交易的永久连接的系统。 根据您是否选择索引所有交易并保留块的完整副本，您可能还需要大量的磁盘空间和RAM。 到2016年底，全索引节点需要2 GB的RAM和125 GB的磁盘空间，以便它有增长的空间。 比特币节点还传输和接收比特币交易和块，消耗互联网带宽。 如果您的互联网连接受限，有带宽上限或按流量计费，建议您不要在其上运行比特币全节点，或以限制其带宽的方式运行它（请参阅资源有限的系统）。

提示 Bitcoin Core默认情况下保留区块链的完整副本，与2009年成立以来在比特币网络上发生的每一笔交易相关。此数据集的大小为120GB，下载可能需要几天或几周，具体取决于CPU和互联网连接的速度。直到完整的块链数据集被下载完成之前，Bitcoin Core将无法处理交易或更新帐户余额。 确保您有足够的磁盘空间，带宽和时间来完成初始同步。 您可以配置Bitcoin Core通过丢弃旧块来减少块链的大小（请参阅资源有限的系统），但是在丢弃数据之前仍将下载整个数据集。

尽管有这些资源需求，但仍有成千上万的志愿者运行比特币节点。 一些在简单的系统上运行，就像树莓派Raspberry Pi（一块35美元的计算机，一张卡的大小）。 许多志愿者还在租用的服务器上运行比特币节点，通常是Linux的一些变体。 虚拟专用服务器（VPS）或云计算服务器实例可用于运行比特币节点。 这些服务器可以从各种供应商每月租用25至50美元。

为什么要运行一个节点？ 以下是一些最常见的原因：

如果您正在开发比特币软件，并且需要依靠比特币节点进行可编程（API）访问网络和区块链。

如果您正在构建必须根据比特币共识规则验证交易的应用程序。 比特币软件公司通常运行几个节点。

如果你想支持比特币。 运行节点使网络更加健壮，能够提供更多的钱包，更多的用户和更多的交易。

如果您不想依赖任何第三方来处理或验证您的交易。

如果您正在阅读本书并对开发比特币软件感兴趣，那么您应该运行自己的节点。

3.2.5首次运行比特币核心

当你第一次运行bitcoind时，它会提醒你用一个安全密码给JSON-RPC接口创建一个配置文件。该密码控制对Bitcoin Core提供的应用程序编程接口（API）的访问。

通过在终端输入bitcoind就可以运行bitcoind了：

```
$ bitcoind
Error: To use the "-server" option, you must set a rpcpassword in the configuration fi
le:
/home/ubuntu/.bitcoin/bitcoin.conf
It is recommended you use the following random password:
rpcuser=bitcoinrpc
rpcpassword=2XA4DuKNCbtZXsBQRRNDEwEY2nM6M4H9Tx5dFjoAVVbK
(you do not need to remember this password)
The username and password MUST NOT be the same.
If the file does not exist, create it with owner-readable-only file permissions.
It is also recommended to set alertnotify so you are notified of problems;
for example: alertnotify=echo %s | mail -s "Bitcoin Alert" admin@foo.com
```

你可以看到，第一次运行bitcoind它会告诉你，你需要建立一个配置文件，至少有一个rpcuser和rpcpassword条目。 另外，建议您设置警报机制。 在下一节中，我们将介绍各种配置选项，并设置一个配置文件。

3.2.6配置比特币核心节点

在首选编辑器中编辑配置文件，并设置参数，用bitcoind推荐的强密码替换密码。 请勿使用本书中显示的密码。 在.bitcoin目录（在用户的主目录下）中创建一个文件，以便它被命名为.bitcoin / bitcoin.conf并提供用户名和密码：

```
rpcuser=bitcoinrpc
rpcpassword=CHANGE_THIS
```

除了rpcuser和rpcpassword选项，Bitcoin Core还提供了100多个配置选项，可以修改网络节点的行为，区块链的存储以及其操作的许多其他方面。

要查看这些选项的列表，请运行bitcoind --help：

```
 bitcoind --help
Bitcoin Core Daemon version v0.11.2

Usage:
  bitcoind [options]                     Start Bitcoin Core Daemon

Options:

  -?
       This help message

  -alerts
       Receive and display P2P network alerts (default: 1)

  -alertnotify=<cmd>
       Execute command when a relevant alert is received or we see a really
       long fork (%s in cmd is replaced by message)
...
[many more options]
...

  -rpcsslciphers=<ciphers>
       Acceptable ciphers (default:
       TLSv1.2+HIGH:TLSv1+HIGH:!SSLv2:!aNULL:!eNULL:!3DES:@STRENGTH)
```

以下是您可以在配置文件中设置的一些最重要的选项，或作为bitcoind的命令行参数：

alertnotify

运行指定的命令或脚本，通常通过电子邮件将紧急警报发送给该节点的所有者。

conf

配置文件的另一个位置。 这只是作为bitcoind的命令行参数有意义，因为它不能在它引用的配置文件内。

datadir

选择要放入所有块链数据的目录和文件系统。 默认情况下，这是您的主目录的.bitcoin子目录。 确保这个文件系统具有几GB的可用空间。

prune

通过删除旧的块，将磁盘空间要求降低到这个兆字节。 在资源受限的节点上不能满足完整块的节点使用这个。

txindex

维护所有交易的索引。 这意味着可以通过ID以编程方式检索任何交易的块链的完整副本。

maxconnections

设置接受连接的最大节点数。 从默认值减少该值将减少您的带宽消耗。 如果您的网络是按照流量计费，请使用。

maxmempool

将交易内存池限制在几兆字节。 使用它来减少节点的内存使用。

maxreceivebuffer/maxsendbuffer

将每连接内存缓冲区限制为1000字节的多个倍数。 在内存受限节点上使用。

minrelaytxfee

设置您将继续的最低费用交易。 低于此值，交易被视为零费用。 在内存受限的节点上使用它来减少内存中交易池的大小。

交易数据库索引和***txindex***选项

默认情况下，Bitcoin Core构建一个仅包含与用户钱包有关的交易的数据库。 如果您想要使用诸如getrawtransaction（参见探索和解码交易)之类的命令访问任何交易，则需要配置Bitcoin Core以构建完整的交易索引，这可以通过txindex选项来实现。 在Bitcoin Core配置文件中设置txindex = 1。 如果不想一开始设置此选项，后期再想设置为完全索引，则需要使用-reindex选项重新启动bitcoind，并等待它重建索引。

下面的完整索引节点的例子配置显示了如何将上述选项与完全索引节点组合起来，作为比特币应用程序的API后端运行。

例3-1完整索引节点的例子

```
alertnotify=myemailscript.sh "Alert: %s"
datadir=/lotsofspace/bitcoin
txindex=1
rpcuser=bitcoinrpc
rpcpassword=CHANGE_THIS
```

下面是小型服务器资源不足配置示例。

例3-2小型服务器资源不足配置示例

```
alertnotify=myemailscript.sh "Alert: %s"
maxconnections=15
prune=5000
minrelaytxfee=0.0001
maxmempool=200
maxreceivebuffer=2500
maxsendbuffer=500
rpcuser=bitcoinrpc
rpcpassword=CHANGE_THIS
```

编辑配置文件并设置最符合您需求的选项后，可以使用此配置测试 bitcoind。 运行Bitcoin Core，使用选项printtoconsole在前台运行输出到控制台：

```
$ bitcoind -printtoconsole

Bitcoin version v0.11.20.0
Using OpenSSL version OpenSSL 1.0.2e 3 Dec 2015
Startup time: 2015-01-02 19:56:17
Using data directory /tmp/bitcoin
Using config file /tmp/bitcoin/bitcoin.conf
Using at most 125 connections (275 file descriptors available)
Using 2 threads for script verification
scheduler thread start
HTTP: creating work queue of depth 16
No rpcpassword set - using random cookie authentication
Generated RPC authentication cookie /tmp/bitcoin/.cookie
HTTP: starting 4 worker threads
Bound to [::]:8333
Bound to 0.0.0.0:8333
Cache configuration:
* Using 2.0MiB for block index database
* Using 32.5MiB for chain state database
* Using 65.5MiB for in-memory UTXO set
init message: Loading block index...
Opening LevelDB in /tmp/bitcoin/blocks/index
Opened LevelDB successfully

[... more startup messages ...]
```

一旦您确信正在加载正确的设置并按预期运行，您可以按Ctrl-C中断进程。要在后台运行Bitcoin Core作为进程，请使用守护程序选项启动它，如bitcoind -daemon。要监视比特币节点的进度和运行状态，请使用命令bitcoin-cli getinfo：

```
$ bitcoin-cli getinfo
{
    "version" : 110200,
    "protocolversion" : 70002,
    "blocks" : 396328,
    "timeoffset" : 0,
    "connections" : 15,
    "proxy" : "",
    "difficulty" : 120033340651.23696899,
    "testnet" : false,
    "relayfee" : 0.00010000,
    "errors" : ""
}
```

这显示运行Bitcoin Core版本0.11.2的节点，块链接高度为396328个块和15个活动网络连接。

一旦您对所选择的配置选项感到满意，您应该将bitcoin添加到操作系统中的启动脚本中，以使其连续运行，并在操作系统重新启动时自动启动。 您可以在contrib / init下的bitcoin的源目录中的各种操作系统和README.md文件中找到一些示例启动脚本，显示哪个系统使用哪个脚本。

3.3 通过命令行使用比特币核心的JSON-RPC API接口

比特币核心客户端实现了JSON-RPC接口，这个接口也可以通过命令行帮助程序bitcoin-cli访问。命令行可以使用API进行编程，让我们有能力进行交互实验。开始前，调用help命令查看可用的比特币RPC命令列表：

```
$ bitcoin-cli help
addmultisigaddress nrequired ["key",...] ( "account" )
addnode "node" "add|remove|onetry"
backupwallet "destination"
createmultisig nrequired ["key",...]
createrawtransaction [{"txid":"id","vout":n},...] {"address":amount,...}
decoderawtransaction "hexstring"
...
...
verifymessage "bitcoinaddress" "signature" "message"
walletlock
walletpassphrase "passphrase" timeout
walletpassphrasechange "oldpassphrase" "newpassphrase"
```

这些命令中的每一个可能需要多个参数。 要获得更多帮助，详细说明和参数信息，请在帮助后添加命令名称。 例如，要查看getblockhash RPC命令的帮助：

```
$ bitcoin-cli help getblockhash
getblockhash index

Returns hash of block in best-block-chain at index provided.

Arguments:
1. index         (numeric, required) The block index

Result:
"hash"         (string) The block hash

Examples:
> bitcoin-cli getblockhash 1000
> curl --user myusername --data-binary '{"jsonrpc": "1.0", "id":"curltest", "method":
"getblockhash", "params": [1000] }' -H 'content-type: text/plain;' http://127.0.0.1:83
32/
```

在帮助信息的最后，您将看到RPC命令的两个示例，使用bitcoin-cli helper或HTTP客户端的curl。 这些例子演示如何调用命令。 复制第一个示例并查看结果：

```
$ bitcoin-cli getblockhash 1000
00000000c937983704a73af28acdec37b049d214adbda81d7e2a3dd146f6ed09
```

结果是一个区块哈希，这在下面的章节中有更详细的描述。 但是现在，该命令应该在您的系统上返回相同的结果，表明您的Bitcoin Core节点正在运行，正在接受命令，并且有关于块1000的信息返回给您。

在下一节中，我们将演示一些非常有用的RPC命令及其预期输出。

3.3.1 获得比特币核心客户端状态的信息

命令： getinfo

比特币 getinfo RPC命令显示关于比特币网络节点、钱包、区块链数据库状态的基础信息。使用 bitcoin-cli 运行它：

```
$ bitcoin-cli getinfo
{
    "version" : 110200,
    "protocolversion" : 70002,
    "blocks" : 396367,
    "timeoffset" : 0,
    "connections" : 15,
    "proxy" : "",
    "difficulty" : 120033340651.23696899,
    "testnet" : false,
    "relayfee" : 0.00010000,
    "errors" : ""
}
```

数据以JavaScript对象表示法（JSON）返回，这是一种格式，可以轻松地被所有编程语言“消费”，但也是非常人性化的。 在这些数据中，我们看到比特币软件客户端（110200）和比特币协议（70002）的版本号。 我们看到当前的块高度，显示了这个客户端知道了多少块（396367）。 我们还会看到有关比特币网络和与此客户端相关的设置的各种统计信息。

提示 比特币特客户端“赶上”当前的blockchain高度需要一些时间，因为它从其他bitcoin客户端下载块。 您可以使用getinfo检查其进度，以查看已知块的数量。

3.3.1 .1探索和解码交易

命令：getrawtransaction，decodeawtransaction

在买咖啡的故事中，Alice从Bob咖啡厅买了一杯咖啡。 她的交易记录在交易ID（txid）0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2的封锁上。 我们使用API通过传递交易ID作为参数来检索和检查该交易：

```
$ bitcoin-cli getrawtransaction 0627052b6f28912f2703066a912ea577f2ce4da4caa5a↵
5fbd8a57286c345c2f2

0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd734d2804fe65fa35779000↵
000008b483045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4↵
ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813014↵
10484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adfffffffff0260e3160000000↵
0001976a914ab68025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef8000000000001976a9↵
147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac00000000
```

提示交易ID在交易被确认之前不具有权威性。 在区块链中缺少交易哈希并不意味着交易未被处理。 这被称为“交易可扩展性”，因为在块中确认之前可以修改交易哈希。 确认后，txid是不可改变的和权威的。

命令getrawtransaction以十六进制返回顺序交易。 为了解码，我们使用decodeawtransaction命令，将十六进制数据作为参数传递。 您可以复制getrawtransaction返回的十六进制，并将其作为参数粘贴到decodeawtransaction中：

```
$ bitcoin-cli decoderawtransaction 0100000001186f9f998a5aa6f048e51dd8419a14d8↵
a0f1a8a2836dd734d2804fe65fa35779000000008b483045022100884d142d86652a3f47ba474↵
6ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298↵
cad530a863ea8f53982c09db8f6e381301410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fd↵
e0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa↵
336a8d752adfffffffff0260e31600000000001976a914ab68025513c3dbd2f7b92a94e0581f5↵
d50f654e788acd0ef8000000000001976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8↵
88ac00000000
```

```
{
  "txid": "0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2",
  "size": 258,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "7957a35fe64f80d234d76d83a2...8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "scriptSig": {
        "asm":"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1decc...",
        "hex":"483045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1de..."
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.01500000,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 ab68...5f654e7 OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a914ab68025513c3dbd2f7b92a94e0581f5d50f654e788ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA"
        ]
      }
    },
    {
      "value": 0.08450000,
      "n": 1,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 7f9b1a...025a8 OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK"
        ]
      }
    }
  ]
}
```

交易解码展示这笔交易的所有成分，包括交易的输入及输出。在这个例子中，我们可以看到这笔给我们新地址存入50mBTC的交易使用了一个输入并且产生两个输出。这笔交易的输入是前一笔确认交易的输出（展示位以d3c7开头的 vin txid）。两个输出则是50mBTC存入额度及返回给发送者的找零。

我们可以使用相同命令（例如 gettransaction ）通过检查由本次交易的txid索引的前一笔交易进一步探索区块链。通过从 一笔交易跳到另外一笔交易，我们可以追溯一连串的交易，因为币值一定是从一个拥有者的地址传送到另一个拥有者的地址。

3.3.2 探索区块

命令： getblock 、 getblockhash

探索区块类似于探索交易。 但是，块可以由块高度或块哈希引用。 首先，让我们找到一个块的高度。 在买咖啡故事中，我们看到Alice的交易已被包含在框277316中。我们使用getblockhash命令，它将块高度作为参数，并返回该块的块哈希值：

```
$ bitcoin-cli getblockhash 277316
0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4
```

既然我们知道我们的交易在哪个区块中，我们可以使用getblock命令，并把区块哈希值作为参数来查询对应的区块：

```
$ bitcoin-cli getblock 0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b3↵
1b2cc7bdc4
```

```
{
  "hash": "000000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4",
  "confirmations": 37371,
  "size": 218629,
  "height": 277316,
  "version": 2,
  "merkleroot": "c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e",
  "tx": [
    "d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f",
    "b268b45c59b39d759614757718b9918caf0ba9d97c56f3b91956ff877c503fbe",
    "04905ff987ddd4cfe603b03cfb7ca50ee81d89d1f8f5f265c38f763eea4a21fd",
    "32467aab5d04f51940075055c2f20bbd1195727c961431bf0aff8443f9710f81",
    "561c5216944e21fa29dd12aaa1a45e3397f9c0d888359cb05e1f79fe73da37bd",
[... hundreds of transactions ...]
    "78b300b2a1d2d9449b58db7bc71c3884d6e0579617e0da4991b9734cef7ab23a",
    "6c87130ec283ab4c2c493b190c20de4b28ff3caf72d16ffa1ce3e96f2069aca9",
    "6f423dbc3636ef193fd8898dfdf7621dcade1bbe509e963ffbff91f696d81a62",
    "802ba8b2adabc5796a9471f25b02ae6aeee2439c679a5c33c4bbcee97e081196",
    "eaaf6a048588d9ad4d1c092539bd571dd8af30635c152a3b0e8b611e67d1a1af",
    "e67abc6bd5e2cac169821afc51b207127f42b92a841e976f9b752157879ba8bd",
    "d38985a6a1bfd35037cb7776b2dc86797abbb7a06630f5d03df2785d50d5a2ac",
    "45ea0a3f6016d2bb90ab92c34a7aac9767671a8a84b9bcce6c019e60197c134b",
    "c098445d748ced5f178ef2ff96f2758cbec9eb32cb0fc65db313bcac1d3bc98f"
  ],
  "time": 1388185914,
  "mediantime": 1388183675,
  "nonce": 924591752,
  "bits": "1903a30c",
  "difficulty": 1180923195.258026,
  "chainwork": "000000000000000000000000000000000000000000000934695e92aaf53afa1a",
  "previousblockhash": "0000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f05
69",
  "nextblockhash": "000000000000000010236c269dd6ed714dd5db39d36b33959079d78dfd431ba7"
}
```


该块包含419笔交易，列出的第64笔交易（0627052b ...）是Alice的咖啡付款。 高度条目告诉我们这是区块链中的第277316块。

3.3.3使用比特币核心的编程接口

bitcoin-cli helper对于探索Bitcoin Core API和测试功能非常有用。 但是应用编程接口的全部要点是以编程方式访问功能。 在本节中，我们将演示从另一个程序访问Bitcoin Core。

Bitcoin Core的API是一个JSON-RPC接口。 JSON代表JavaScript对象符号，它是一种非常方便的方式来呈现人类和程序可以轻松阅读的数据。 RPC代表远程过程调用，这意味着我们通过网络协议调用远程（位于比特币核心节点）的过程（函数）。 在这种情况下，网络协议是HTTP或HTTPS（用于加密连接）。

当我们使用bitcoin-cli命令获取命令的帮助时，它给了我们一个例子，它使用curl，通用的命令行HTTP客户端来构造这些JSON-RPC调用之一：

```
$ curl --user myusername --data-binary '{"jsonrpc": "1.0", "id":"curltest", "method":
"getinfo", "params": [] }' -H 'content-type: text/plain;' http://127.0.0.1:8332/
```

此命令显示curl向本地主机（127.0.0.1）提交HTTP请求，连接到默认比特币端口（8332），并使用text / plain编码向getinfo方法提交jsonrpc请求。

如果您在自己的程序中实现JSON-RPC调用，则可以使用通用的HTTP库构建调用，类似于前面的curl示例所示。

然而，大多数编程语言中都使用库，以“包装”比特币核心API的方式使其简单得多。 我们将使用python-bitcoinlib库来简化API访问。 请记住，这需要您有一个运行的Bitcoin Core实例，将用于进行JSON-RPC调用。

下面的例子通运行getinfo中的Python脚本进行简单的getinfo调用，并从Bitcoin Core返回的数据中打印区块参数。

例3-3通过Bitcoin Core的JSON-RPC API运行

```
link:code/rpc_example.py[]
```

运行结果如下：

```
$ python rpc_example.py
394075
```

它告诉我们，我们的本地Bitcoin Core节点在其块链中有394075个块。 这不是一个惊人的结果，但它演示了使用库作为Bitcoin Core的JSON-RPC API的简化接口的基本使用。

接下来，我们使用getrawtransaction和decodetransaction调用来检索Alice咖啡付款的详细信息。 在下面的例子中，我们检索Alice的交易并列出交易的输出。 对于每个输出，我们显示收件人地址和值。 作为提醒，Alice的交易有一个输出支付Bob的咖啡馆和一个输出找回Alice。

例3-4检索交易并迭代其输出

```
link:code/rpc_transaction.py[]
```

运行结果如下：

```
$ python rpc_transaction.py
([u'1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA'], Decimal('0.01500000'))
([u'1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK'], Decimal('0.08450000'))
```

上述两个例子都比较简单。 你眞的不需要一个程序来运行它们; 你可以很容易地使用bitcoin-cli helper。 然而，下一个示例需要数百个RPC调用，并更清楚地说明了使用编程接口的方便。

在时，我们首先检索块277316，然后通过引用每个交易ID来检索每个419个交易。 接下来，我们迭代每个交易的输出并将其加起来。例3-5检索块并添加所有交易输出

```
link:code/rpc_block.py[]
```

运行结果如下：

```
$ python rpc_block.py

('Total value in block: ', Decimal('10322.07722534'))
```

我们的示例代码计算出，此块中交易的总价值为10,322.07722534 个BTC（包括25 BTC奖励和0.0909 BTC费用）。 通过搜索块哈希或高度来比较区块浏览器站点报告的数量。 一些区块浏览器报告不包括奖励的总价值，不包括交易费用。 看看是否可以发现差异。

3.4 其他替代客户端、资料库、工具包

除了参考客户端（bitcoind），还可以使用其他的客户端和资料库去连接比特币网络和数据结构。这些工具都由一系列 的编程语言执行，用他们各自的语言为比特币程序提供原生的交互。以下列出了一部分由编程语言组织的一些最好的库，客户端和工具包：

C/C++

Bitcoin Core The reference implementation of bitcoin

libbitcoin Cross-platform C++ development toolkit, node, and consensus library

bitcoin explorer Libbitcoin’s command-line tool

picocoin A C language lightweight client library for bitcoin by Jeff Garzik

JavaScript

bcoin A modular and scalable full-node implementation with API

Bitcore Full node, API, and library by Bitpay

BitcoinJS A pure JavaScript Bitcoin library for node.js and browsers

Java

bitcoinj A Java full-node client library

Bits of Proof (BOP) A Java enterprise-class implementation of bitcoin

Python

python-bitcoinlib A Python bitcoin library, consensus library, and node by Peter Todd

pycoin A Python bitcoin library by Richard Kiss

pybitcointools A Python bitcoin library by Vitalik Buterin

Ruby

bitcoin-client A Ruby library wrapper for the JSON-RPC API

Go

btcd A Go language full-node bitcoin client

Rust

rust-bitcoin Rust bitcoin library for serialization, parsing, and API calls

C#

NBitcoin Comprehensive bitcoin library for the .NET framework

Objective-C

CoreBitcoin Bitcoin toolkit for ObjC and Swift

更多的库存在各种其他编程语言，也会一直更新的。

你可能听说过比特币是基于密码学，这一在计算机安全中广泛使用的数学分支。 密码学在希腊语中是“秘密写作”的意思，但密码学这门科学不仅只包含被称之为秘密写作的加密学。 密码学也可以用来证明秘密的知识，而不会泄露秘密（数字签名），或证明数据的真实性（数字指纹）。 这些类型的加密证明是比特币中关键的数学工具并在比特币应用程序中被广泛使用。具有讽刺意味的是，加密不是比特币的重要组成部分，因为它的通信和交易数据没有加密，也不需要加密来保护资金。 在本章中，我们将介绍一些在比特币中用来控制资金的所有权的密码学，包括密钥，地址和钱包。

4.1 简介

比特币的所有权是通过数字密钥、比特币地址和数字签名来确定的。数字密钥实际上并不存储在网络中，而是由用户生成之后，存储在一个叫做钱包的文件或简单的数据库中。存储在用户钱包中的数字密钥完全独立于比特币协议，可由用户的钱包软件生成并管理，而无需参照区块链或访问网络。密钥实现了比特币的许多有趣特性，包括去中心化信任和控制、所 有权认证和基于密码学证明的安全模型。

大多数比特币交易都需要一个有效的签名才会被存储在区块链。只有有效的密钥才能产生有效的数字签名，因此拥有~密钥副本就拥有了对该帐户的比特币的控制权。用于支出资金的数字签名也称为见证（witness），密码术中使用的术语。 比特币交易中的见证数据证明了所用资金的真正归谁所有。

密钥是成对出现的，由一个私钥和一个公钥所组成。公钥就像银行的帐号，而私钥就像控制账户的PIN码或支票的签名。比特币的用户很少会直接看到数字密钥。一般情况下，它们被存储在钱包文件内，由比特币钱包软件进行管理。

在比特币交易的支付环节，收件人的公钥是通过其数字指纹代表的，称为比特币地址，就像支票上的支付对象的名字 （即“收款方”）。一般情况下，比特币地址由一个公钥生成并对应于这个公钥。然而，并非所有比特币地址都是公钥； 他们也可以代表其他支付对象，譬如脚本，我们将在本章后面提及。这样一来，比特币地址把收款方抽象起来了，使得 交易的目的地更灵活，就像支票一样：这个支付工具可支付到个人账户、公司账户，进行账单支付或现金支付。比特币地址是用户经常看到的密钥的唯一代表，他们只需要把比特币地址告诉其他人即可。

首先，我们将介绍密码学并解释在比特币中使用的数学知识。然后我们将了解密钥如何被产生、存储和管理。我们将回顾用于代表私钥和公 钥、地址和脚本地址的各种编码格式。最后，我们将讲解密钥和地址的高级用途：比特币靓号，多重签名以及脚本地址和纸钱包。

4.1.1 公钥加密和加密货币

公钥加密发明于20世纪70年代。它是计算机和信息安全的数学基础。

自从公钥加密被发明之后，一些合适的数学函数被发现，譬如：素数幂和椭圆曲线乘法。这些数学函数都是不可逆的， 就是说很容易向一个方向计算，但不可以向相反方向倒推。基于这些数学函数的密码学，使得生成数字密钥和不可伪造 的数字签名成为可能。比特币正是使用椭圆曲线乘法作为其公钥加密的基础。

在比特币系统中，我们用公钥加密创建一个密钥对，用于控制比特币的获取。密钥对包括一个私钥，和由其衍生出的唯 一的公钥。公钥用于接收比特币，而私钥用于比特币支付时的交易签名。

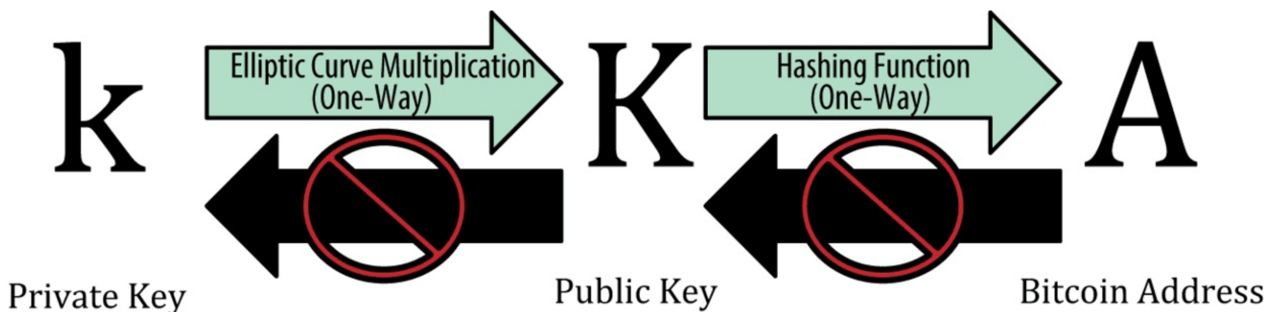
公钥和私钥之间的数学关系，使得私钥可用于生成特定消息的签名。此签名可以在不泄露私钥的同时对公钥进行验证。

支付比特币时，比特币的当前所有者需要在交易中提交其公钥和签名（每次交易的签名都不同，但均从同一个私钥生 成）。比特币网络中的所有人都可以通过所提交的公钥和签名进行验证，并确认该交易是否有效，即确认支付者在该时刻对所交易的比特币拥有所有权。

提示 大多数比特币钱包工具为了方便会将私钥和公钥以密钥对的形式存储在一起。然而，公钥可以由私钥计算得到， 所以只存储私钥也是可以的。

4.1.2 私钥和公钥

一个比特币钱包中包含一系列的密钥对，每个密钥对包括一个私钥和一个公钥。私钥（k）是一个数字，通常是随机选出的。有了私钥，我们就可以使用椭圆曲线乘法这个单向加密函数产生一个公钥（K）。有了公钥（K），我们就可以使 用一个单向加密哈希函数生成比特币地址（A）。在本节中，我们将从生成私钥开始，讲述如何使用椭圆曲线运算将私 钥生成公钥，并最终由公钥生成比特币地址。私钥、公钥和比特币地址之间的关系如下图所示。



为什么使用非对称加密（公钥/私钥）？

为什么在比特币中使用非对称密码术？ 它不是用于“加密”（make secret）交易。 相反，非对称密码学的有用属性是生成数字签名的能力。 可以将私钥应用于交易的数字指纹以产生数字签名。 该签名只能由知晓私钥的人生成。 但是，访问公钥和交易指纹的任何人都可以使用它们来验证签名。 这种非对称密码学的适用性使得任何人都可以验证每笔交易的每个签名，同时确保只有私钥的所有者可以产生有效的签名。

4.1.3 私钥

私钥就是一个随机选出的数字而已。一个比特币地址中的所有资金的控制取决于相应私钥的所有权和控制权。在比特币交易中，私钥用于生成支付比特币所必需的签名以证明对资金的所有权。私钥必须始终保持机密，因为一旦被泄露给第三 方，相当于该私钥保护之下的比特币也拱手相让了。私钥还必须进行备份，以防意外丢失，因为私钥一旦丢失就难以复原，其所保护的比特币也将永远丢失。

提示 比特币私钥只是一个数字。你可以用硬币、铅笔和纸来随机生成你的私钥：掷硬币256次，用纸和笔记录正反面并转换为0和1，随机得到的256位二进制数字可作为比特币钱包的私钥。该私钥可进一步生成公钥。

从一个随机数生成私钥 生成密钥的第一步也是最重要的一步，是要找到足够安全的熵源，即随机性来源。生成一个比特币私钥在本质上与“在1 到2^256之间选一个数字”无异。只要选取的结果是不可预测或不可重复的，那么选取数字的具体方法并不重要。比特币软件使用操作系统底层的随机数生成器来产生256位的熵（随机性）。通常情况下，操作系统随机数生成器由人工的随机源进行初始化，这就是为什么也可能需要不停晃动鼠标几秒钟。

更准确地说，私钥可以是1和n-1之间的任何数字，其中n是一个常数（n=1.158 * 10^77，略小于2^256），并被定义为由比特币所使用的椭圆曲线的阶（见椭圆曲线密码学解释）。要生成这样的一个私钥，我们随机选择一个256位的数字，并检查它是否小于n-1。从编程的角度来看，一般是通过在一个密码学安全的随机源中取出一长串随机字节，对其使用SHA256哈希算法进行运算，这样就可以方便地产生一个256位的数字。如果运算结果小于n-1，我们就有了一个合适的私钥。否则，我们就用另一个随机数再重复一次。

警告 不要自己写代码或使用你的编程语言提供的简易随机数生成器来获得一个随机数。使用密码学安全的伪随机数生成器（CSPRNG），并且需要有一个来自具有足够熵值的源的种子。使用随机数发生器的程序库时，需仔细研读其文档，以确保它是加密安全的。正确实施CSPRNG是密钥安全性的关键所在。

以下是一个随机生成的私钥（k），以十六进制格式表示（256位的二进制数，以64位十六进制数显示，每个十六进制数占4位）：

1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD

提示比特币私钥空间的大小是2^256，这是一个非常大的数字。用十进制表示的话，大约是10^77，而可见宇宙被估计只含有10^80个原子。

要使用比特币核心客户端生成一个新的密钥，可使用 getnewaddress 命令。出于安全考虑，命令运行后只 显示生成的公钥，而不显示私钥。如果要bitcoind显示私钥，可以使用 dumpprivkey 命令。 dumpprivkey 命令会把私钥以 Base58校验和编码格式显示，这种私钥格式被称为钱包导入格式（WIF，Wallet Import Format），在“私钥的格式”一节有详细讲解。下面给出了使用这两个命令生成和显示私钥的例子：

$ bitcoin-cli getnewaddress 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy $ bitcoin-cli dumpprivkey 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

dumpprivkey 命令打开钱包提取由 getnewaddress 命令生成的私钥。除非密钥对都存储在钱包里，否则bitcoind的并不能从公钥得知私钥。 dumpprivkey 命令才有效。

提示 dumpprivkey命令无法从公钥得到对应的私钥，因为这是不可能的。这个命令只是显示钱包中已有也就是由getnewaddress命令生成的私钥。

您还可以使用Bitcoin Explorer命令行工具（请参阅附录中的[appdx_bx]）使用命令seed，ec-new和ec-to-wif生成和显示私钥：

$ bx seed | bx ec-new | bx ec-to-wif 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn

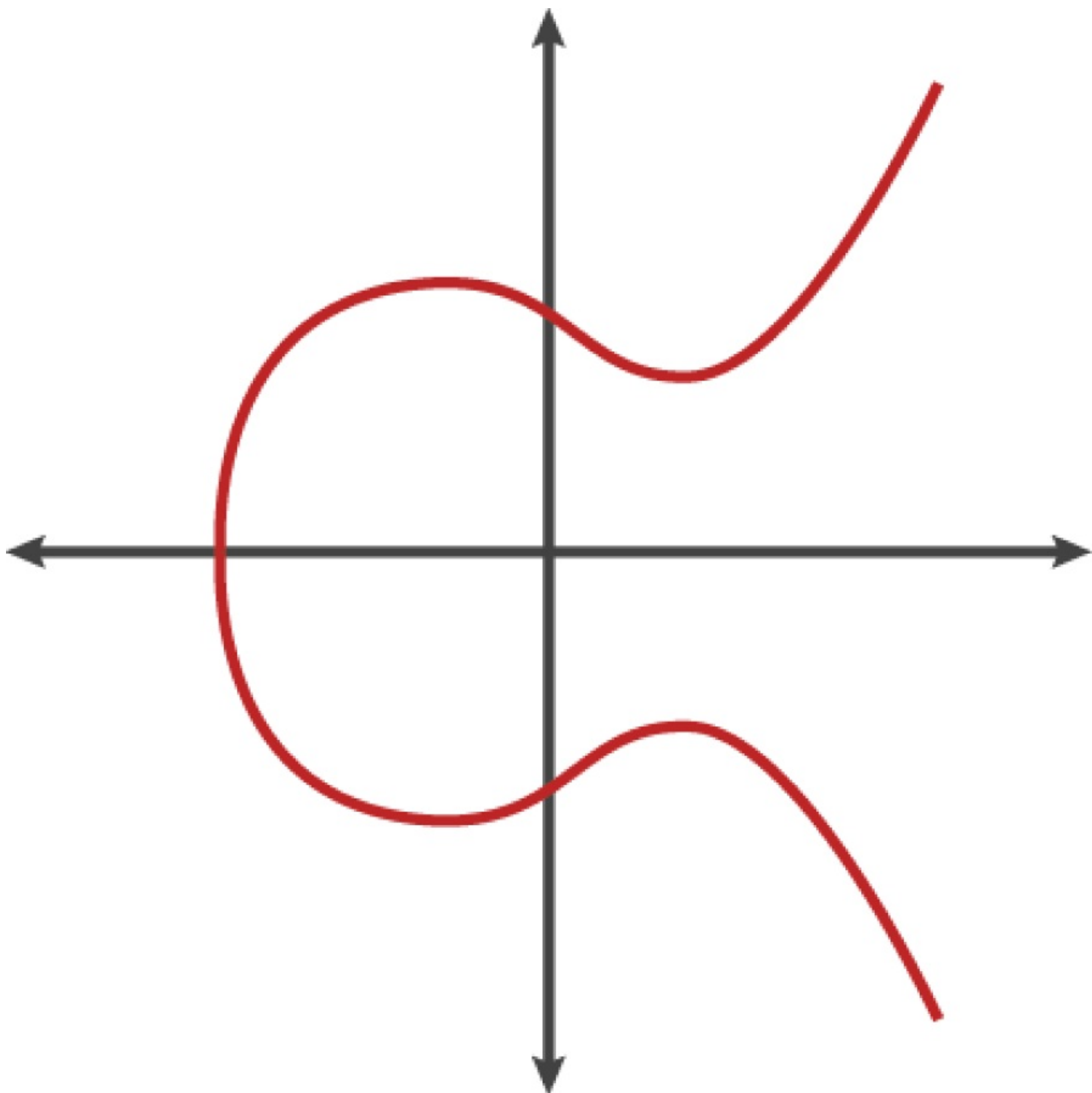
4.1.4 公钥

通过椭圆曲线乘法可以从私钥计算得到公钥，这是不可逆转的过程：K = k * G 。其中k是私钥，G是被称为生成点的常数点，而K是所得公钥。其反向运算，被称为“寻找离散对数”——已知公钥K来求出私钥k——是非常困难的，就像去试验所有可能的k值，即暴力搜索。在演示如何从私钥生成公钥之前，我们先稍微详细学习下椭圆曲线密码学。

提示 椭圆曲线乘法是密码学家称之为“陷阱门”功能的一种函数：在一个方向（乘法）很容易做，而不可能在相反的方向（除法）做。 私钥的所有者可以容易地创建公钥，然后与世界共享，知道没有人可以从公钥中反转函数并计算出私钥。 这个数学技巧成为证明比特币资金所有权的不可伪造和安全的数字签名的基础。

4.1.5 椭圆曲线密码学（Elliptic Curve Cryptography）解释

椭圆曲线加密法是一种基于离散对数问题的非对称加密法，可以用对椭圆曲线上的点进行加法或乘法运算来表达。 下图是一个椭圆曲线的示例，类似于比特币所用的曲线。



比特币使用了secp256k1标准所定义的一种特殊的椭圆曲线和一系列数学常数。该标准由美国国家标准与技术研究院（NIST）设立。secp256k1曲线由下述函数定义，该函数可产生一条椭圆曲线：

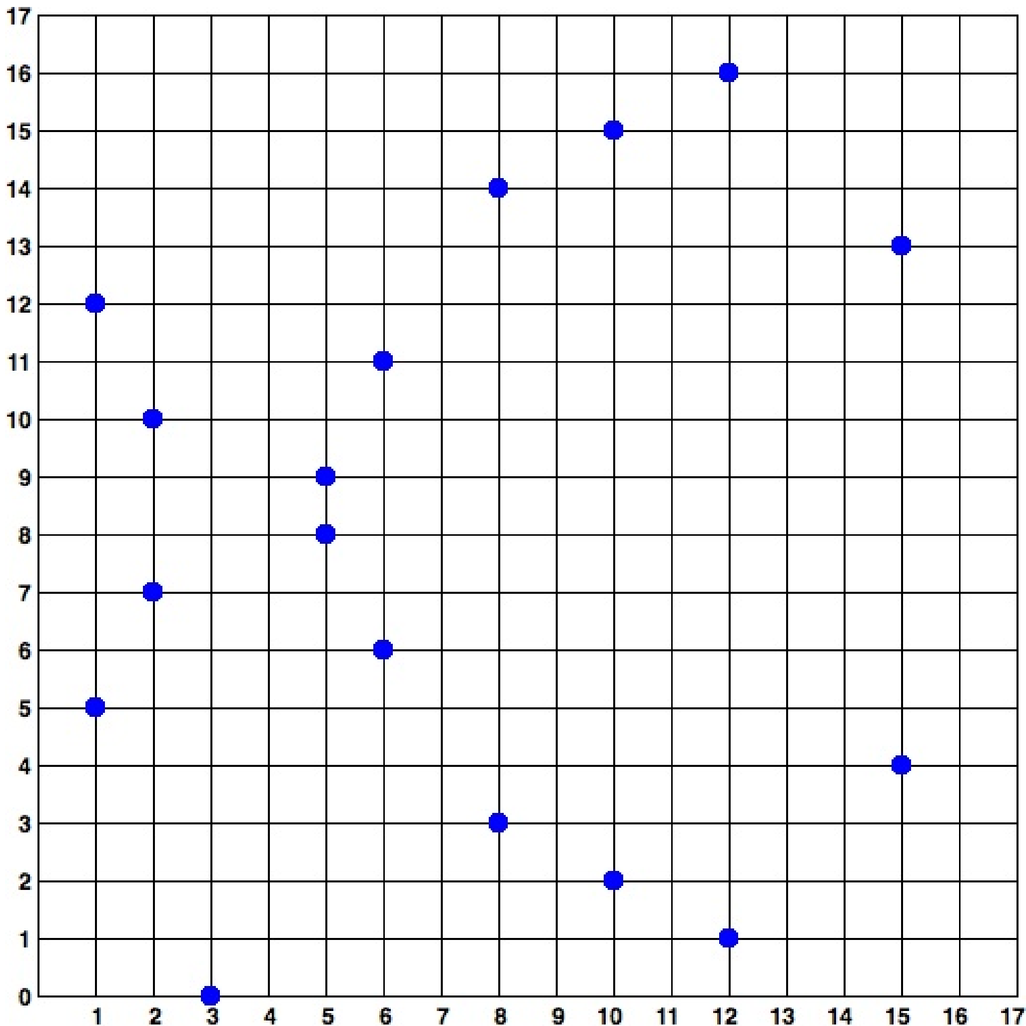
$y^2 = (x^3 + 7)\}$ over (F_p)

或

$y^2 \bmod p = (x^3 + 7) \bmod p$

上述mod p（素数p取模）表明该曲线是在素数阶p的有限域内，也写作Fp，其中p = 2^256 – 2^32 – 2^9 – 2^8 – 2^7 – 2^6 – 2^4 – 1， 这是个非常大的素数。 因为这条曲线被定义在一个素数阶的有限域内，而不是定义在实数范围，它的函数图像看起来像分散在两个维度上的散落的点，因此很难可视化。不过，其中的数学原理与实数范围的椭圆曲线相似。作为一个例

子，下图显示了在一个小了很多的素数阶17的有限域内的椭圆曲线，其形式为网格上的一系列散点。而secp256k1的比特币椭圆曲线可以被想象成一个极大的网格上一系列更为复杂的散点。



下面举一个例子，这是 secp256k1 曲线上的点P，其坐标为(x，y)。可以使用Python对其检验：

```
P = (55066263022277343669578718895168534326250603453777594175500187360389116729240,
32670510020758816978083085130507043184471273380659243275938904335757337482424)
```

在椭圆曲线的数学原理中，有一个点被称为“无穷远点”，这大致对应于0在加法中的作用。计算机中，它有时表示为X = Y = 0（虽然这不满足椭圆曲线方程，但可作为特殊情况进行检验）。

还有一个 + 运算符，被称为“加法”，就像小学数学中的实数相加。给定椭圆曲线上的两个点P1和P2，则椭圆曲线上必定有第三点 P3 = P1 + P2。 几何图形中，该第三点P3可以在P1和P2之间画一条线来确定。这条直线恰好与椭圆曲线相交于另外一个地方。此点记为 P3'= (x，y)。然后，在x轴做翻折获得 P3=(x，-y)。

下面是几个可以解释“穷远点”之存在需要的特殊情况。

若 P1和 P2是同一点，P1和P2间的连线则为点P1 的切线。曲线上有且只有一个新的点与该切线相交。该切线的斜率可用微积分求得。即使限制曲线点为两个整数坐标也可求得斜率！

在某些情况下（即，如果P1和P2具有相同的x值，但不同的y值），则切线会完全垂直，在这种情况下，P3 = “无穷远点”。

若P1就是“无穷远点”，那么其和 P1 + P2= P2。类似地，当P2是无穷远点，则P1+ P2 = P1。这就是把无穷远点类似于0的作用。 事实证明，在这里 + 运算符遵守结合律，这意味着 (A+B)+C = A+(B+C)。这就是说我们可以直接不加括号书写 A + B + C，而不至于混淆。 因此，我们已经定义了椭圆加法，我们可以对乘法用拓展加法的标准方法进行定义。给定椭圆曲线上的点P，如果k是整数，则 kP = P + P + P + …+ P（k次）。注意，在这种情况下k有时被混淆而称为“指数”。

4.1.6 生成公钥

以一个随机生成的私钥k为起点，我们将其与曲线上预定的生成点G相乘以获得曲线上的另一点，也就是相应的公钥 K。生成点是secp256k1标准的一部分，比特币密钥的生成点都是相同的：

{K = k * G}

其中k是私钥，G是生成点，在该曲线上所得的点K是公钥。因为所有比特币用户的生成点是相同的，一个私钥k乘以G将 得到相同的公钥K。k和K之间的关系是固定的，但只能单向运算，即从k得到K。这就是可以把比特币地址（K的衍生） 与任何人共享而不会泄露私钥（k）的原因。

提示 因为其中的数学运算是单向的，所以私钥可以转换为公钥，但公钥不能转换回私钥。

为实现椭圆曲线乘法，我们 以之前产生的私钥k和与生成点G相乘得到公钥K：

K =
1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD * G

公钥K 被定义为一个点 K = (x, y)：

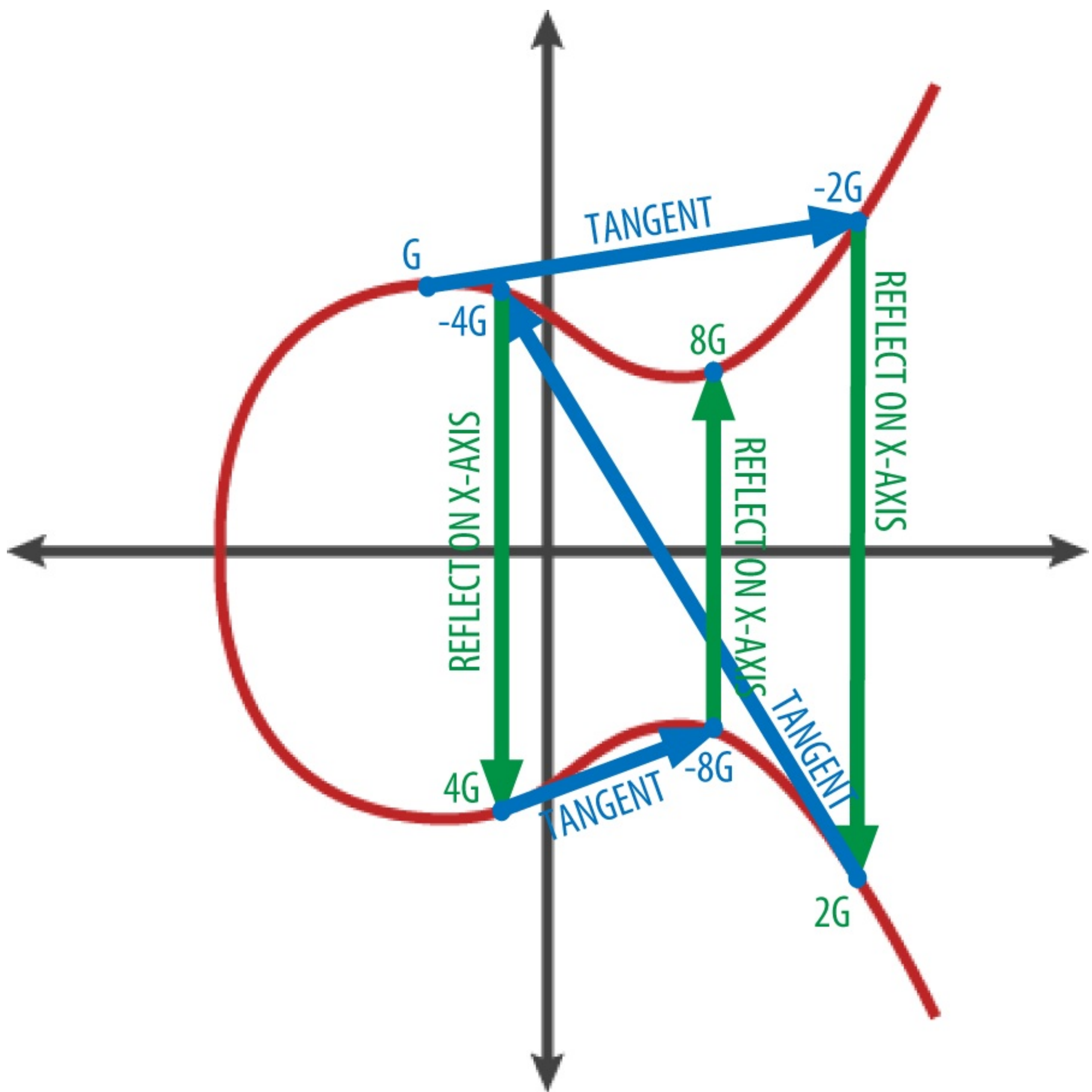
K = (x, y)

其中，

x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB

为了展示整数点的乘法，我们将使用较为简单的实数范围的椭圆曲线。请记住，其中的数学原理是相同的。我们的目标是找到生成点G的倍数kG。也就是将G相加k次。在椭圆曲线中，点的相加等同于从该点画切线找到与曲线相交的另一 点，然后翻折到x轴。

下图显示了在曲线上得到 G、2G、4G 的几何操作。



提示大多数比特币程序使用OpenSSL加密库进行椭圆曲线计算。例如，调用EC_POINT_mul() 函数，可计算得到公钥。

4.2 比特币地址

比特币地址是一个由数字和字母组成的字符串，可以与任何想给你比特币的人分享。由公钥（一个同样由数字和字母组 成的字符串）生成的比特币地址以数字“1”开头。下面是一个比特币地址的例子：

1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy

在交易中，比特币地址通常以收款方出现。如果把比特币交易比作一张支票，比特币地址就是收款人，也就是我们要写入收款人一栏的内容。一张支票的 收款人可能是某个银行账户，也可能是某个公司、机构，甚至是现金支票。支票不需要指定一个特定的账户，而是用一个

抽象的名字作为收款人，这使它成为一种相当灵活的支付工具。与此类似，比特币地址使用类似的抽象，也使比特币交易变得很灵活。比特币地址可以代表一对公钥和私钥的所有者，也可以代表其 它东西，比如会在后面的“P2SH (Pay-to-Script-Hash)”一节讲到的付款脚本。现在，让我们来看一个简单的例子，由公钥生成比特币地址。

比特币地址可由公钥经过单向的加密哈希算法得到。哈希算法是一种单向函数，接收任意长度的输入产生指纹或哈希。加密哈希函数在比特币中被广泛使用 ：比特币地址、脚本地址以及在挖矿中的工作量证明算法。由公钥生成比特币地址时使用的算法是Secure Hash Algorithm (SHA)和the RACE Integ rity Primitives Evaluation Message Digest (RIPEMD)，具体地说是SHA256和RIPEMD160。

以公钥 K 为输入，计算其SHA256哈希值，并以此结果计算RIPEMD160 哈希值，得到一个长度为160位（20字节）的数字：

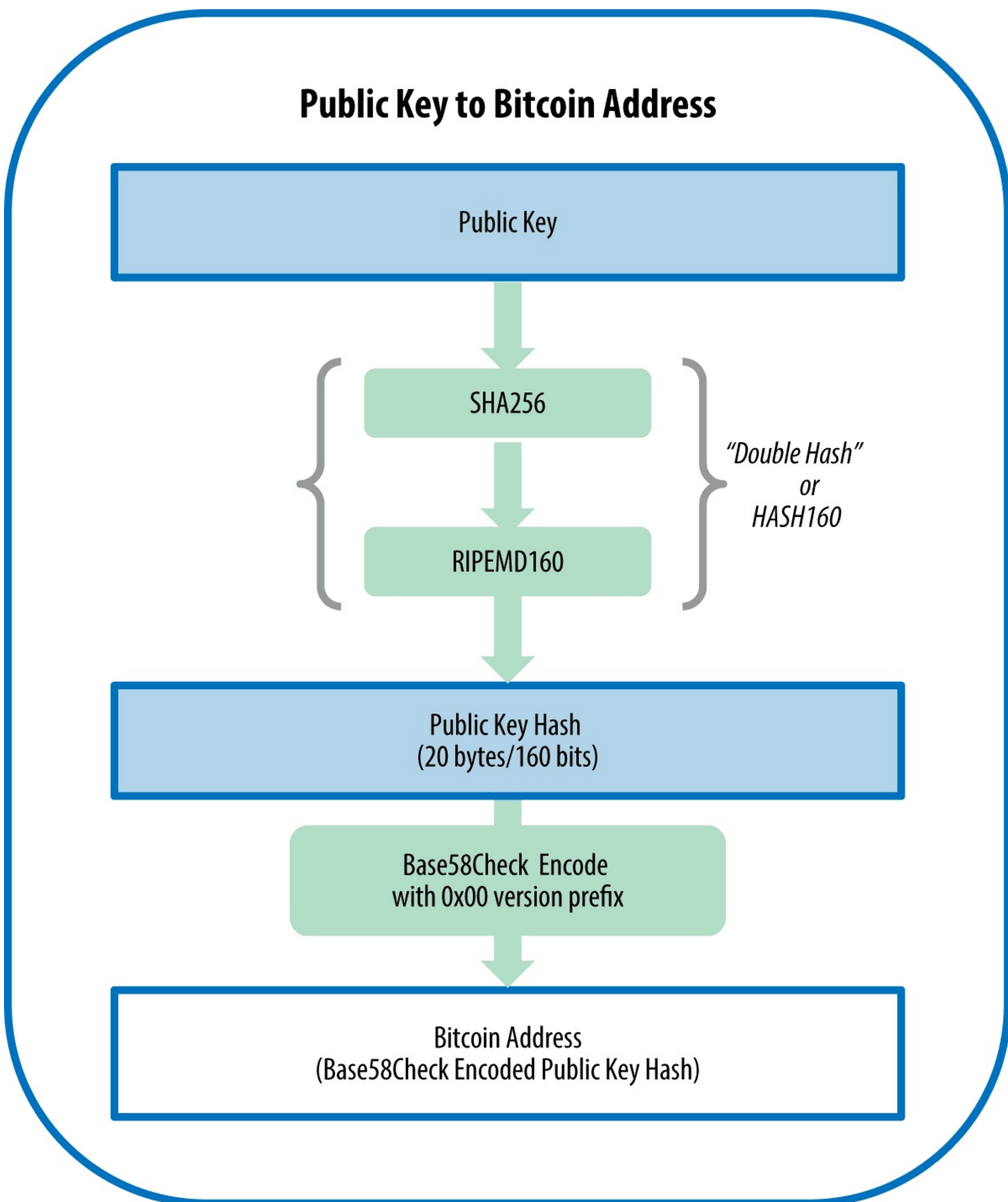
A = RIPEMD160(SHA256(K))

公式中，K是公钥，A是生成的比特币地址。

提示比特币地址与公钥不同。比特币地址是由公钥经过单向的哈希函数生成的。

通常用户见到的比特币地址是经过“Base58Check”编码的（参见“Base58和Base58Check编码”一节），这种编码 使用了58个字符（一种Base58数字系统）和校验码，提高了可读性、避免歧义并有效防止了在地址转录和输入中产生 的错误。Base58Check编码也被用于比特币的其它地方，例如比特币地址、私钥、加密的密钥和脚本哈希中，用来提高可读性和录入的正确性。下一节中我们会详细解释Base58Check的编码和解码机制，以及它产生的结果。

下图描述了如何从公钥生成比特币地址。



4.2.1 Base58和Base58Check编码

为了更简洁方便地表示长串的数字，使用更少的符号，许多计算机系统会使用一种以数字和字母组成的大于十进制的表示法。例如，传统的十进制计数系统使用0-9十个数字，而十六进制系统使用了额外的 A-F 六个字母。一个同样的数字，它的十六进制表 示就会比十进制表示更短。甚至更加简洁，Base64使用了26个小写字母、26个大写字母、10个数字以及两个符号（例 如"+"和"/"），用于在电子邮件这样的基于文本的媒介中传输二进制数据。Base64通常用于编码邮件中的附件。Base58 是一种基于文本的二进制编码格式，用在比特币和其它的加密

货币中。这种编码格式不仅实现了数据压缩，保持了易读 性，还具有错误诊断功能。Base58是Base64编码格式的子集，同样使用大小写字母和10个数字，但舍弃了一些容易错 读和在特定字体中容易混淆的字符。具体地，Base58不含Base64中的0（数字0）、O（大写字母o）、l（小写字母 L）、I（大写字母i），以及“+”和“/”两个字符。简而言之，Base58就是由不包括（0，O，l，I）的大小写字母和数字组成。

例4-1 比特币的Base58字母表

123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz

为了增加防止打印和转录错误的安全性，Base58Check是一种常用在比特币中的Base58编码格式，比特币有内置的检查错误的编码。检验和是添加到正在编码的数据末端的额外4个字节。校验和是从编码的数据的哈希值中得到的，所以可以用来检测并避免转录和输入中产生的错误。使用Base58check编码时，解码软件会计算数据的校验和并和编码中自带的校验和进行对比。二者不匹配则表明有错误产生，那么这个Base58Check的数据就是无效的。一个错误比特币地址就不会被钱包软件认为是有效的地址，否则这种错误会造成资金的丢失。

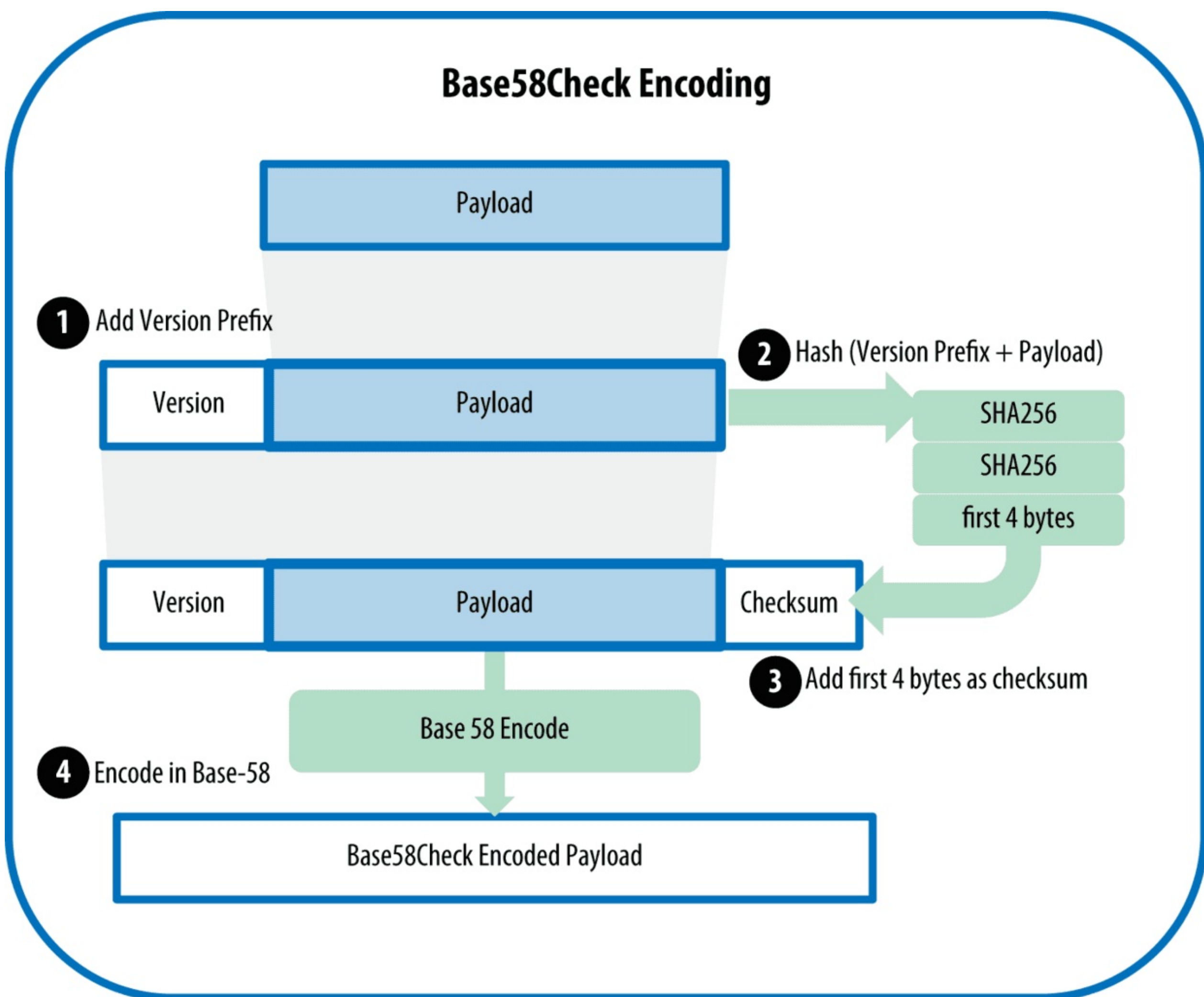
为了将数据（数字）转换成Base58Check格式，首先我们要对数据添加一个称作“版本字节”的前缀，这个前缀用来识别编码的数据的类 型。例如，比特币地址的前缀是0（十六进制是0x00），而对私钥编码时前缀是128（十六进制是0x80）。 表4-1会列出一些常见版本的前缀。

接下来，我们计算“双哈希”校验和，意味着要对之前的结果（前缀和数据）运行两次SHA256哈希算法：

checksum = SHA256(SHA256(prefix+data))

在产生的长32个字节的哈希值（两次哈希运算）中，我们只取前4个字节。这4个字节就作为检验错误的代码或者校验和。校验码会添加到数据之后。

结果由三部分组成：前缀、数据和校验和。这个结果采用之前描述的Base58字母表编码。下图描述了Base58Check编 码的过程。



在比特币中，大多数需要向用户展示的数据都使用Base58Check编码，可以实现数据压缩，易读而且有错误检验。 Base58Check编码中的版本前缀是用来创造易于辨别的格式，在Base58里的格式在Base58Check编码的有效载荷的开始包含了明确的属性。这些属性使用户可以轻松明确被编码的数据的类型以及如何使用它们。例如我们可以看到他们的不同，Base58Check编码的比特币地址是以1开头的，而Base58Check编码的私钥WIF是以5开头的。表4-1展示了一些版本前缀和他们对应的Base58格式。

表4-1 Base58Check版本前缀和编码后的结果

Type	Version prefix (hex)	Base58 result prefix
Bitcoin Address	0x00	1
Pay-to-Script-Hash Address	0x05	3
Bitcoin Testnet Address	0x6F	m or n
Private Key WIF	0x80	5, K, or L
BIP-38 Encrypted Private Key	0x0142	6P
BIP-32 Extended Public Key	0x0488B21E	xpub

我们回顾比特币地址产生的完整过程，从私钥、到公钥（椭圆曲线上某个点）、再到两次哈希的地址，到最终的 Base58Check编码。例4-3的C++代码完整详细的展示了从私钥到Base58Check编码后的比特币地址的 步骤。代码中使用“3.3 其他客户端、资料库、工具包”一节中介绍的libbitcoin library来实现某些辅助功能。

例4-3.从私钥中创建Base58Check编码的比特币地址

link:code/addr.cpp[]

代码使用预定义的私钥在每次运行时产生相同的比特币地址，如下例所示

例4-3.编译并运行addr代码

Compile the addr.cpp code $ g++ -o addr addr.cpp $(pkg-config --cflags --libs libbitcoin) Run the addr executable $ ./addr Public key: 0202a406624211f2abbdc68da3df929f938c3399dd79fac1b51b0e4ad1d26a47aa Address: 1PRTTaJesdNovgne6Ehcdu1fpEdX7913CK

4.2.2 密钥的格式

公钥和私钥的都可以有多种格式。一个密钥被不同的格式编码后，虽然结果看起来可能不同，但是密钥所编码数字并没有改变。这些不同的编码格式主要是用来方便人们无误地使用和识别密钥。

4.2.2.1私钥的格式

私钥可以以许多不同的格式表示，所有这些都对应于相同的256位的数字。表4-2展示了私钥的三种常见格式。不同的格式用在不同的场景下。十六进制和原始的二进制格式用在软件的内部，很少展示给用户看。WIF格式用在钱包之间密钥的输入和输出，也用于代表私钥的二维码（条形码）。

Type	Prefix	Description
Raw	None	32 bytes
Hex	None	64 hexadecimal digits
WIF	5	Base58Check encoding: Base58 with version prefix of 128- and 32-bit checksum
WIF-compressed	K or L	As above, with added suffix 0x01 before encoding

表4-3 示例：同样的私钥，不同的格式

Format	Private key
Hex	1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
WIF-compressed	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

这些表示法都是用来表示相同的数字、相同的私钥的不同方法。虽然编码后的字符串看起来不同，但不同的格式彼此之 间可以很容易地相互转换。请注意，“raw binary”未显示在表4-3示例中，根据定义此处显示的任何编码的格式，不是raw binary数据。

我们使用Bitcoin Explorer中的wif-to-ec命令（请参阅[appdx_bx]）来显示两个WIF键代表相同的私钥：

$ bx wif-to-ec 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn

1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd

$ bx wif-to-ec KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd

4.2.3从Base58Check解码

Bitcoin Explorer命令（参见[appdx_bx]）使得编写shell脚本和命令行“管道”变得容易，这些方式可以处理比特币密钥，地址和交易。 您可以使用Bitcoin Explorer在命令行上解码Base58Check格式。

我们使用base58check-decode命令解码未压缩的密钥：

$ bx base58check-decode
5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn

wrapper

{

```
checksum 4286807748
payload 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd
version 128
```

}

结果包含密钥作为有效载荷，WIF版本前缀128和校验和。

请注意，压缩密钥的“有效负载”附加了后缀01，表示导出的公钥要压缩：

$ bx base58check-decode
KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

wrapper {

```
checksum 2339607926
payload 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd01
version 128
```

}

将十六进制转换为***Base58Check***编码

要转换成Base58Check（与上一个命令相反），我们使用Bitcoin Explorer的base58check-encode命令（请参阅[appdx_bx]），并提供十六进制私钥，其次是WIF版本前缀128：

bx base58check-encode
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd --version 128

5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn

将十六进制（压缩格式密钥）转换为***Base58Check***编码

要将压缩格式的私钥编码为Base58Check（参见“压缩格式私钥”一节），我们需在十六进制私钥的后面添加后缀01，然后使用跟上面一样的方法：

$ bx base58check-encode
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd01 --version 128

KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

生成的WIF压缩格式的私钥以字母“K”开头，用以表明被编码的私钥有一个后缀“01”，且该私钥只能被用于生成压缩格式 的公钥（参见“压缩格式公钥”一节）。

4.2.3.1公钥的格式

公钥也可以用多种不同格式来表示，最重要的是它们分为非压缩格式或压缩格式公钥这两种形式。

我们从前文可知，公钥是在椭圆曲线上的一个点，由一对坐标（x，y）组成。公钥通常表示为前缀04紧接着两个256比特的数字。其中一个256比特数字是公钥的x坐标，另一个256比特数字是y坐标。前缀04是用来区分非压缩格式公钥， 压缩格式公钥是以02或者03开头。

下面是由前文中的私钥所生成的公钥，其坐标x和y如下：

x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A

y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB

下面是同样的公钥以520比特的数字（130个十六进制数字）来表达。这个520比特的数字以前缀04开头，紧接着是x及y 坐标，组成格式为04 x y：

K =
04F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A0
7CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE 52DDFE2E505BDB

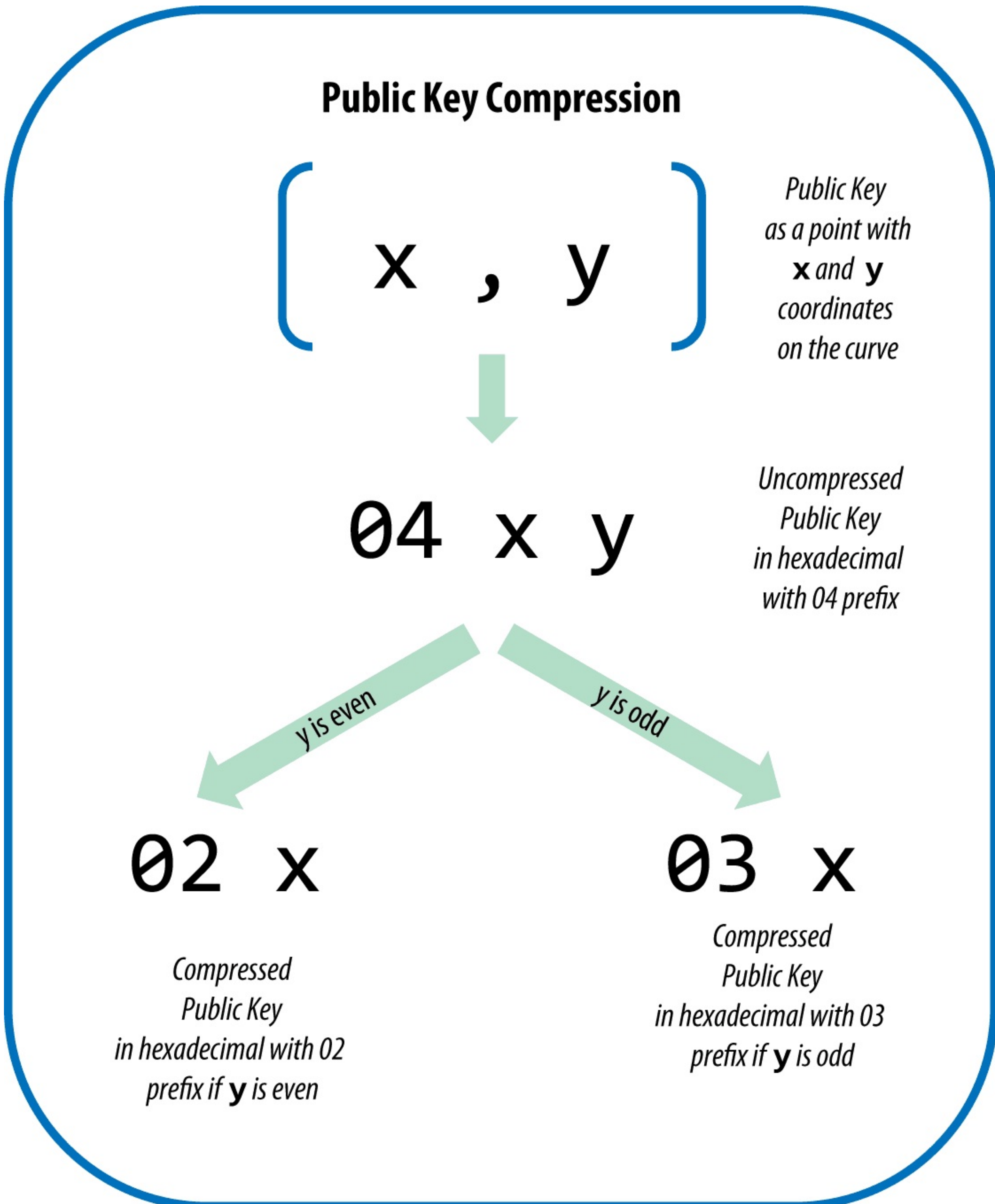
4.2.3.2压缩格式公钥

引入压缩格式公钥是为了减少比特币交易的字节数，从而可以节省那些运行区块链数据库的节点磁盘空间。大部分比特币交易包含了公钥，用于验证用户的凭据和支付比特币。每个公钥有520比特（包括前缀，x坐标，y坐标）。如果每个区块有数百个交易，每天有成千上万的交易发生，区块链里就会被写入大量的数据。

正如我们在“4.1.4 公钥”一节所见，一个公钥是一个椭圆曲线上的点(x, y)。而椭圆曲线实际是一个数学方程，曲线上的点实际是该方程的一个解。因此，如果我们知道了公钥的x坐标，就可以通过解方程y2 mod p = (x3 + 7) mod p得到y坐 标。这种方案可以让我们只存储公钥的x坐标，略去y坐标，从而将公钥的大小和存储空间减少了256比特。每个交易所 需要的字节数减少了近一半，随着时间推移，就大大节省了很多数据传输和存储。

未压缩格式公钥使用04作为前缀，而压缩格式公钥是以02或03作为前缀。需要这两种不同前缀的原因是：因为椭圆曲 线加密的公式的左边是y2 ，也就是说y的解是来自于一个平方根，可能是正值也可能是负值。更形象地说，y坐标可能在 x坐标轴的上面或者下面。从图4-2的椭圆曲线图中可以看出，曲线是对称的，从x轴看就像对称的镜子两面。因此，如果我们略去y坐标，就必须储存y的符号（正值或者负值）。换句话说，对于给定的x值，我们需要知道y值在x轴的上面还是下面，因为它们代表椭圆曲线上不同的点，即不同的公钥。当我们在素数p

阶的有限域上使用二进制算术计算椭圆曲线的时候，y坐标可能是奇数或者偶数，分别对应前面所讲的y值的正负符号。因此，为了区分y坐标的两种可能值，我们在生成压缩格式公钥时，如果y是偶数，则使用02作为前缀；如果y是奇数，则使用03作为前缀。这样就可以根据公钥中给定的x值，正确推导出对应的y坐标，从而将公钥解压缩为在椭圆曲线上的完整的点坐标。下图阐释了公钥压缩：



下面是前述章节所生成的公钥，使用了264比特（66个十六进制数字）的压缩格式公钥格式，其中前缀03表示y坐标是一个奇数：

K =
03F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A

这个压缩格式公钥对应着同样的一个私钥，这意味它是由同样的私钥所生成。但是压缩格式公钥和非压缩格式公钥看起来不同。更重要的是，如果我们使用双哈希函数(RIPEMD160(SHA256(K)))将压缩格式公钥转化成比特币地址，得到的地址将会不同于由非压缩格式公钥产生的地址。这种结果会让人迷惑，因为一个私钥可以生成两种不同格式的公钥——压缩格式和非压缩格式，而这两种格式的公钥可以生成两个不同的比特币地址。但是，这两个不同的比特币地址的私钥是一样的。

压缩格式公钥渐渐成为了各种不同的比特币客户端的默认格式，它可以大大减少交易所需的字节数，同时也让存储区块链所需的磁盘空间变小。然而，并非所有的客户端都支持压缩格式公钥，于是那些较新的支持压缩格式公钥的客户端就不得不考虑如何处理那些来自较老的不支持压缩格式公钥的客户端的交易。这在钱包应用导入另一个钱包应用的私钥的 时候就会变得尤其重要，因为新钱包需要扫描区块链并找到所有与这些被导入私钥相关的交易。比特币钱包应该扫描哪个比特币地址呢？新客户端不知道应该使用哪个公钥：因为不论是通过压缩的公钥产生的比特币地址，还是通过非压缩的公钥产生的地址，两个都是合法的比特币地址，都可以被私钥签名，但是他们是不同的比特币地址。

为了解决这个问题，当私钥从钱包中被导出时，代表私钥的WIF在较新的比特币钱包里被处理的方式不同，表明该私钥已经被用来生成压缩的公钥和因此压缩的比特币地址。这个方案可以解决导入私钥来自于老钱包还是新钱包的问题，同时也解决了通过公钥生成的比特币地址是来自于压缩格式公钥还是非压缩格式公钥的问题。最后新钱包在扫描区块链时，就可以使用对应的比特币地址去查找该比特币地址在区块链里所发生的交易。我们将在下一节详细解释这种机制是如何工作的。

4.2.3.3压缩格式私钥

实际上“压缩格式私钥”是一种名称上的误导，因为当一个私钥被使用WIF压缩格式导出时，不但没有压缩，而且比“非压缩格式”私钥长出一个字节。这个多出来的一个字节是私钥被加了后缀01，用以表明该私钥是来自于一个较新的钱包， 只能被用来生成压缩的公钥。私钥是非压缩的，也不能被压缩。“压缩的私钥”实际上只是表示“用于生成压缩格式公钥的私钥”，而“非压缩格式私钥”用来表明“用于生成非压缩格式公钥的私钥”。为避免更多误解，应该只可以说导出格式 是“WIF压缩格式”或者“WIF”，而不能说这个私钥是“压缩”的。

表4示例：相同的密钥，不同的格式

Format	Private key
Hex	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
Hex-compressed	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD01
WIF-compressed	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

请注意，十六进制压缩私钥格式在末尾有一个额外的字节（十六进制为01）。 虽然Base58编码版本前缀对于WIF和WIF压缩格式都是相同的（0x80），但在数字末尾添加一个字节会导致Base58编码的第一个字符从5变为K或 L，考虑到对于Base58这是十进制编码100号和99号之间的差别。对于100是一个数字长于99的数字，它有一个前缀1，而不是前缀9。当长度变化，它会影响前缀。 在Base58中，前缀5改变为K或L，因为数字的长度增加一个字节。

要注意的是，这些格式并不是可互换使用的。在实现了压缩格式公钥的较新的钱包中，私钥只能且永远被导出为WIF压 缩格式（以K或L为前缀）。对于较老的没有实现压缩格式公钥的钱包，私钥将只能被导出为WIF格式（以5为前缀）导 出。这样做的目的就是为了给导入这些私钥的钱包一个信号：是否钱包必须搜索区块链寻找压缩或非压缩公钥和地址。

如果一个比特币钱包实现了压缩格式公钥，那么它将会在所有交易中使用该压格式缩公钥。钱包中的私钥将会被用来在曲线上生成公钥点，这个公钥点将会被压缩。压缩格式公钥然后被用来生成交易中使用的比特币地址。当从一个实现了压缩格式公钥的新的比特币钱包导出私钥时，钱包导入格式（WIF）将会被修改为WIF压缩格式，该格式将会在私钥的后面附加一个字节大小的后缀01。最 终的Base58Check编码格式的私钥被称作WIF（“压缩”）私钥，以字母“K”或“L”开头。而以“5”开头的是从较老的钱包中 以WIF（非压缩）格式导出的私钥。

提示 “压缩格式私钥”是一个不当用词！私钥不是压缩的。WIF压缩格式的私钥只是用来表明他们只能被生成压缩的公钥和对应的比特币地址。相反地，“WIF压缩”编码的私钥还多出一个字节，因为这种私钥多了后缀“01”。该后缀是用来区分“非压缩格式”私钥和“压缩格式”私钥。

4.3 用Python实现密钥和比特币地址

最全面的比特币Python库是 Vitalik Buterin写的 pybitcointools。在例4-5中，我们使用pybitcointools库（导入 为“bitcoin”）来生成和显示不同格式的密钥和比特币地址。

例4-5 使用pybitcointools库的密钥和比特币地址的生成和格式化过

link:code/key-to-address-ecc-example.py[]

例4-6 上例输出如下：

$ python key-to-address-ecc-example.py

Private Key (hex) is:

3aba4162c7251c891207b747840551a71939b0de081f85c4e44cf7c13e41daa6

Private Key (decimal) is:

26563230048437957592232553826663696440606756685920117476832299673293013768870

Private Key (WIF) is:

5JG9hT3beGTJuUAmCQEmNaxAuMacCTfXuw1R3FCXig23RQHMr4K

Private Key Compressed (hex) is:

3aba4162c7251c891207b747840551a71939b0de081f85c4e44cf7c13e41daa601

Private Key (WIF-Compressed) is:

KyBsPXxTuVD82av65KZkrGrWi5qLMah5SdNq6uftawDbgKa2wv6S

Public Key (x,y) coordinates is:

(41637322786646325214887832269588396900663353932545912953362782457239403430124L,
16388935128781238405526710466724741593761085120864331449066658622400339362166L)

Public Key (hex) is:

045c0de3b9c8ab18dd04e3511243ec2952002dbfadc864b9628910169d9b9b00ec↵
243bcefdd4347074d44bd7356d6a53c495737dd96295e2a9374bf5f02ebfc176

Compressed Public Key (hex) is:

025c0de3b9c8ab18dd04e3511243ec2952002dbfadc864b9628910169d9b9b00ec

Bitcoin Address (b58check) is:

1thMirt546nngXqyPEz532S8fLwbozud8

Compressed Bitcoin Address (b58check) is: 14cxpo3MBCYYWCgF74SWTdcmxipnGUsPw3

例4-7是另外一个示例，使用的是Python ECDSA库来做椭圆曲线计算而非使用bitcoin的库。

link:code/ec-math.py[]

例4-8是上述脚本的输出。

注意： \ Install Python PIP package manager

$ sudo apt-get install python-pip

\Install the Python ECDSA library

$ sudo pip install ecdsa

\ Run the script

$ python ec-math.py

Secret:
38090835015954358862481132628887443905906204995912378278060168703580660294000 EC point:
(70048853531867179489857750497606966272382583471322935454624595540007269312627,
105262206478686743191060800263479589329920209527285803935736021686045542353380)

BTC public key:
029ade3effb0a67d5c8609850d797366af428f4a0d5194cb221d807770a1522873

4.4 高级密钥和地址

在以下部分中，我们将看到高级形式的密钥和地址，诸如加密私钥、脚本和多重签名地址，靓号地址，和纸钱包。

4.4.1 加密私钥（BIP0038）

私钥必须保密。私钥的机密性需求情况是，在实践中相当难以实现，因为该需求与同样重要的安全对象可用性相互矛盾。当你需要为了避免私钥丢失而存储备份时，会发现维护私钥私密性是一件相当困难的事情。通过密码加密存有私钥的钱包可能要安全一点，但那个钱包也需要备份。有时，例如用户因为要升级或重装钱包软件，而需要把密钥从一个钱包转移到另一个。私钥备份也可能需要存储在纸张上（参见"后面纸钱包"一节）或者外部存储介质里，比如U盘。但如果一旦备份文件失窃或丢失呢？这些矛盾的安全目标推进了便携、方便、可以被众多不同钱包和比特币客户端理解的加密私钥标准BIP0038的出台（BIP-38详细可参见附录部分）。

BIP0038提出了一个通用标准，使用一个口令加密私钥并使用Base58Check对加密的私钥进行编码，这样加密的私钥就可以安全地保存在备份介质里，安全地在钱包间传输，保持密钥在任何可能被暴露情况下的安全性。这个加密标准使 用了AES，这个标准由NIST建立，并广泛应用于商业和军事应用的数据加密。

BIP0038加密方案是：输入一个比特币私钥，通常使用WIF编码过，base58chek字符串的前缀“5”。此外BIP0038加密方案需要一个长密码作为口令，通常由多个单词或一段复杂的数字字母字符串组成。BIP0038加密方案的结果是一个由 base58check编码过的加密私钥，前缀为6P。如果你看到一个6P开头的的密钥，这就意味着该密钥是被加密过，并需要一个口令来转换（解码）该密钥回到可被用在任何钱包WIF格式的私钥（前缀为5）。许多钱包APP现在能够识别 BIP0038加密过的私钥，会要求用户提供口令解码并导入密钥。第三方APP，诸如非常好用基于浏览器的Bit Address ， 可以被用来解码BIP00038的密钥。

最通常使用BIP0038加密的密钥用例是纸钱包——一张纸张上备份私钥。只要用户选择了强口令，使用BIP0038加密的私钥的纸钱包就无比的安全，这也是一种很棒的比特币离线存储方式（也被称作“冷存储”）。

在bitaddress.org上测试表4-5中加密密钥，看看如何输入密码以得到加密密钥。

表4-5 BIP0038加密私钥例子

Private Key (WIF)	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
Passphrase	MyTestPassphrase

4.4.2 P2SH (Pay-to-Script Hash)和多重签名地址

正如我们所知，传统的比特币地址从数字1开头，来源于公钥，而公钥来源于私钥。虽然任何人都可以将比特币发送到 一个1开头的地址，但比特币只能在通过相应的私钥签名和公钥哈希值后才能消费。

以数字3开头的比特币地址是P2SH地址，有时被错误的称谓多重签名或多重签名地址。他们指定比特币交易中受益人为哈希的脚本，而不是公钥的所有者。这个特性在2012年1月由BIP0016引进，目前因为BIP0016提供了增加功能到地址本身的机会而被广泛的采纳。不同于P2PKH交易发送资金到传统1开头的比特币地址，资金被发送到3开头的地址时，需要的不仅仅是一个公钥的哈希值和一个私钥签名作为所有者证明。在创建地址的时候，这些要求会被指定在脚本中，所有对地址的输入都会被这些要求阻隔。

一个P2SH地址从交易脚本中创建，它定义谁能消耗这个交易输出（后面“P2SH（Pay-to-Script-Hash）”一节对此有 详细的介绍）。编码一个P2SH地址涉及使用一个在创建比特币地址用到过的双重哈希函数，并且只能应用在脚本而不是公钥：

script hash = RIPEMD160(SHA256(script))

产生的脚本哈希由Base58Check编码前缀为5的版本、编码后得到开头为3的编码地址。一个P2SH地址例子是 3F6i6kwkevjR7AsAd4te2YB2zZyASEm1HM。可以使用Bitcoin Explorer命令脚本编码获得，比如sha256, ripemd160, and base58check-encode，举例如下：

$ echo dup hash160 [89abcdefabbaabbaabbaabbaabbaabbaabbaabba] equalverify checksig > script

$ bx script-encode < script | bx sha256 | bx ripemd160 | bx base58check-encode --version 5

3F6i6kwkevjR7AsAd4te2YB2zZyASEm1HM

提示P2SH 不一定是多重签名的交易。虽然P2SH地址通常都是代表多重签名，但也可能是编码其他类型的交易脚本。

4.4.2.1 多重签名地址和P2SH

目前，P2SH函数最常见的实现是多重签名地址脚本。顾名思义，底层脚本需要多个签名来证明所有权，此后才能消费资金。设计比特币多重签名特性是需要从总共N个密钥中需要M个签名（也被称为“阈值”），被称为M-N多签名，其 中M是等于或小于N。例如，第一章中提到的咖啡店主Bob使用多重签名地址需要1-2签名，一个是属于他的密钥和一个属于他同伴的密钥，以确保其中一方可以签署消费一笔锁定到这个地址的输出。这类似于传统的银行中的一个“联合账户”，其中任何一方配偶可以单独签单消费。或就像Bob雇佣的网页设计师Gopesh， 创立一个网站，可能为他的业务需要一个2-3的多签名地址，确保除非至少两个业务合作伙伴签署签名交易才可以进行支付消费。

我们将会在第五章节探索如何使用P2SH地址创建交易用来消费资金。

4.4.3 比特币靓号地址

靓号地址包含了人类可读信息的有效比特币地址。例如，1LoveBPzzD72PUXLzCkYAtGFYmK5vYNR33就是包含了Base-58 字母love的。靓号地址需要生成并通过数十亿的候选私钥测试，直到一个私钥能生成具有所需图案的比特币地址。虽然有一些优化过的靓号生成算法，该方法必须涉及随机上选择一个私钥，生成公钥，再生成比特币地址，并检查是否与所要的靓号图案相匹配，重复数十亿次，直到找到一个匹配。

一旦找到一个匹配所要图案的靓号地址，来自这个靓号地址的私钥可以和其他地址相同的方式被拥有者消费比特币。靓号地址不比其他地址具有更多或更少的安全性。它们依靠和其他地址相同的ECC和SHA。你无法比任何别的地址更容易的获得一个靓号图案开头的地址的私钥。

在第一章中，我们介绍了Eugenia，一位在菲律宾工作的儿童慈善总监。我们假设Eugenia组织了一场比特币募捐活动，并希望使用靓号比特币地址来宣传这个募捐活动。Eugenia将会创造一个以1Kids开头的靓号地址来促进儿童慈善募捐的活动。让我们看看这个靓号地址如何被创建，这个靓号地址对Eugenia慈善募捐的安全性又意味着什么。

4.4.3.1 生成靓号地址

认识到比特币地址不过是由Base58字母代表的一个数字是非常重要的。搜索“1kids”开头的图案我们会发 现从1Kids11111111111111111111111111111到1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzzzz的地址。这些以“1kid”开头的地址范围中大约有58的29次方地址（1.4 * 10^51)）。表4-6显示了这些有“1kids”前缀的地址。

表4-6 “1Kids”靓号的范围

From	1Kids11111111111111111111111111111
	1Kids11111111111111111111111111112
	1Kids11111111111111111111111111113
	...
To	1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzzzz

我们把“1Kids”这个前缀当作数字，我们可以看看比特币地址中这个前缀出现的频率。如果是一台普通性能的桌面电脑， 没有任何特殊的硬件，可以每秒搜索大约10万个密钥。

Length	Pattern	Frequency	Average search time
1	1K	1 in 58 keys	< 1 milliseconds
2	1Ki	1 in 3,364	50 milliseconds
3	1Kid	1 in 195,000	< 2 seconds
4	1Kids	1 in 11 million	1 minute
5	1KidsC	1 in 656 million	1 hour
6	1KidsCh	1 in 38 billion	2 days
7	1KidsCha	1 in 2.2 trillion	3–4 months
8	1KidsChar	1 in 128 trillion	13–18 years
9	1KidsChari	1 in 7 quadrillion	800 years
10	1KidsCharit	1 in 400 quadrillion	46,000 years
11	1KidsCharity	1 in 23 quintillion	2.5 million years

正如你所见，Eugenia将不会很快地创建出以“1KidsCharity”开头的靓号地址，即使她有数千台的电脑同时进行运算。每增加一个字符就会增加58倍的计算难度。超过七个字符的图案通常需要专用的硬件才能被找出，譬如用户定制的具有多个图形处理单元（GPU）的台式机。那些通常是无法继续在比特币挖矿中盈利的钻机，被重新赋予了寻找靓号地 址的任务。用GPU系统搜索靓号的速度比用通用CPU要快很多个量级。

另一种寻找靓号地址的方法是将工作外包给一个矿池里的靓号矿工们，如靓号矿池中的矿池。一个矿池是一种允许那些 GPU硬件通过为他人寻找靓号地址来获得比特币的服务。对小额的账单，Eugenia可以将搜索7位字符图案的靓号地址的工作外包，在几个小时内就可以得到结果，而不必用一个CPU搜索上几个月才得到结果。

生成一个靓号地址是一项通过蛮力的过程：尝试一个随机密钥，检查生成的地址是否和所需的图案相匹配，重复这个过程直到成功找到为止。例4-9是个靓号矿工的例子，用C++程序写的来寻找靓号地址的程序。这个例子运用到了我们在“其他替代客户端、资料库、工具包”一节介绍过的libbitcoin库。

例4-9 靓号挖掘程序

link:code/vanity-miner.cpp[]

注释编译和运行虚拟矿工示例使用std :: random_device（译者注：std :: random_device是均匀分布的整数随机数生成器，产生非确定性随机数）。 根据实施情况，可能会反映底层操作系统提供的CSRNG。 在类似Unix的操作系统（如Linux）中，它来自/ dev/urandom。 这里使用的随机数字生成器用于演示，并不适用于生成级别的比特币密钥，因为它没有以足够的安全性。

示例代码需要用C编译器链接libbitcoin库（此库需要提前装入该系统）进行编译。可以不带参数直接执行vanity-miner的可执行文件 （参见例4-10），它就会尝试找到以“1kid”开头的靓号地址。

例4-10编译并运行vanity-miner程序示例

\Compile the code with g++ $ g++ -o vanity-miner vanity-miner.cpp

$(pkg-config --cflags --libs libbitcoin)

\Run the example

$ ./vanity-miner

Found vanity address! 1KiDzkG4MxmovZryZRj8tK81oQRhbZ46YT Secret: 57cc268a05f83a23ac9d930bc8565bac4e277055f4794cbd1a39e5e71c038f3f

\ Run it again for a different result

$ ./vanity-miner

Found vanity address! 1Kidxr3wsmMzzouwXibKfwTYs5Pau8TUFn Secret: 7f65bbbbe6d8caae74a0c6a0d2d7b5c6663d71b60337299a1a2cf34c04b2a623

使用时间命令查看需要多久才能找到结果

$ time ./vanity-miner Found vanity address! 1KidPWhKgGRQWD5PP5TAnGfDyfWp5yceXM Secret: 2a802e7a53d8aa237cd059377b616d2bfcfa4b0140bc85fa008f2d3d4b225349

real 0m8.868s user 0m8.828s sys 0m0.035s

正如我们运行Unix命令所测出的运行时间所示，示例代码要花几秒钟来找出匹配“kid”三个字符模板的结果。你可以尝试在源代码中改变search这一搜索模板，看一看如果是四个字符或者五个字符的搜索模板需要花多久时间！

4.4.3.2 靓号地址安全性

靓号地址既可以增加、也可以削弱安全措施，它们着实是一把双刃剑。用于改善安全性时，一个独特的地址使对手难以使用他们自己的地址替代你的地址，以欺骗你的顾客支付他们的账单。不幸的是，靓号地址也可能使得任何人都能创建一个类似于随机地址的地址，甚至另一个靓号地址，从而欺骗你的客户。

Eugenia可以让捐款人捐款到她宣布的一个随机生成地址（例如：1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy）。或者她可以生成一个以“1Kids”开头的靓号地址以显得更独特。

在这两种情况下，使用单一固定地址（而不是每比捐款用一个独立的动态地址）的风险之一是小偷有可能会黑进你的网站，用他自己的地址取代你的地址，从而将捐赠转移给他自己。如果你在不同的地方公布了你的捐款地址，你的用户可以 在付款之前用自己眼睛检查以确保这个地址跟在你的网站、邮件和传单上看到的地址是同一个。在随机地址1j7mdg5rbqyuhenydx39wvwk7fslpeoxzy的情况下，普通用户可能会只检查头几个字符“1j7mdg”，就认为地址匹配。使用靓号地址生成器，那些想通过替换类似地址来盗窃的人可以快速生成与前几个字符相匹配的地址，如表4-8所示。

表4-8 生成匹配某随机地址的多个靓号

Original Random Address	1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
Vanity (4-character match)	1J7md1QqU4LpctBetHS2ZoyLV5d6dShhEy
Vanity (5-character match)	1J7mdgYqyNd4ya3UEcq31Q7sqRMXw2XZ6n
Vanity (6-character match)	1J7mdg5WxGENmwyJP9xuGhG5KRzu99BBCX

那靓号地址会不会增加安全性？如果Eugenia生成1Kids33q44erFfpeXrmDSz7zEqG2FesZEN的靓号地址，用户可能看到靓号图案的字母和一些字符在上面，例如在地址部分中注明了1Kids33。这样就会迫使攻击者生成至少6个字母相匹 配的的靓号地址（比之前多2个字符），就要花费比Eugenia多3364倍的努力。本质上，Eugenia付出的努力（或者靓号池付出的）迫使攻击者不得不生成更长的靓号图案。如果Eugenia花钱请矿池生成8个字符的靓号地址，攻击者将会被逼迫到10字符的境地，那将是个人电脑，甚至昂贵自定义靓号挖掘机或靓号池也无法生成。对Eugenia来说可承担的起支出，对攻击者来说则变成了无法承担支出，特别是如果欺诈的潜在回报不足以支付生成靓号地址所需的费用。

4.4.4 纸钱包

纸钱包是打印在纸张上的比特币私钥。有时纸钱包为了方便起见也包括对应的比特币地址，但这并不是必要的，因为地址可以从私钥中导出。纸钱包是一个非常有效的建立备份或者线下存储比特币（即冷存储）的方式。作为备份机制，一个纸钱包可以提供安全性，以防在电

脑硬盘损坏、失窃或意外删除的情况下造成密钥的的丢失。作为一个冷存储的机制，如果纸钱包密钥在线下生成并永久不在电脑系统中存储，他们在应对黑客攻击，键盘记录器，或其他在线电脑威胁更有安全性。

纸钱包有许多不同的形状，大小，和外观设计，但非常基本的原则是一个密钥和一个地址打印在纸张上。表4-14展现了纸钱包最基本的形式。

表4-9 比特币纸钱包的私钥和公钥的打印形式

Public address	Private key (WIF)
1424C2F4bC9JidNjjTUZCbUxv6Sa1Mt62x	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn

通过使用工具，就可以很容易地生成纸钱包，譬如使用bitaddress.org网站上的客户端Javascript生成器。这个页面包含所有生成密钥和纸钱包所必须代码，甚至在完全失去网络连接的情况下，也可以生成密钥和纸钱包。若要使用它，先将HTML页面保存在本地磁盘或外部U盘。从Internet网络断开，从浏览器中打开文件。更方便的，使用一个原始操作系统启动电脑，比如一 个光盘启动的Linux系统。任何在脱机情况下使用这个工具所生成的密钥，都可以通过USB线在本地打印机上打印出 来，从而制造了密钥只存在纸张上而从未存储在在线系统上的纸钱包。将这些纸钱包放置在防火保险柜内，发送比特币到 对应的比特币地址上，从而实现了一个简单但非常有效的冷存储解决方案。图4-8展示了通过bitaddress.org 生成的纸钱包。



这个简单的纸钱包系统的不足之处是那些被打印下来的密钥容易被盗窃。一个能够接近这些纸的小偷只需偷走纸或者用把拍摄纸上的密钥，就能控制被这些密钥锁定的比特币。一个更复杂的纸钱包存储系统使用BIP0038加密的私钥。打印在纸钱包上的这些私钥被其所有者记住的一个口令保护起来。没有口令，这些被加密过的密钥也是毫无用处的。但它们仍旧优于用口令保护，因为这些密钥从没有在线过，并且必须从保险箱或者其他物理的安全存储中导出。图4-9展示了通过bitaddress.org 生成的加密纸钱包。



警告虽然你可以多次存款到纸钱包中，但是你最好一次性提取里面所有的资金。因为如果你提取的金额少于其中的总金额的话，有些钱包可能会生成一个找零地址。并且，你所用的电脑可能被病毒感染，那么就有可能泄露私钥。一 次性提走所有余款可以减少私钥泄露的风险，如果你所需的金额比较少，那么请把余额发送到相同交易的一个新的纸钱包里。

纸钱包有许多设计和大小，并有许多不同的特性。有些作为礼物送给他人，有季节性的主题，像圣诞节和新年主题。另 外一些则是设计保存在银行金库或通过某种方式隐藏私钥的保险箱内，或者用不透明的刮刮贴，或者折叠和防篡改的铝箔胶粘密封。图4-10至图4-12展示了几个不同安全和备份功能的纸钱包的例子。

图4-10 通过bitcoinpaperwallet.com生成的、私钥写在折叠袋上的纸钱包

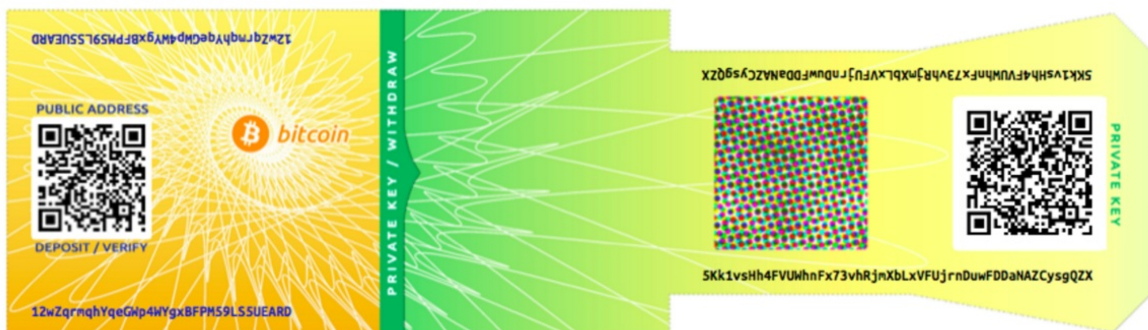


图4-11 通过bitcoinpaperwallet.com 生成的、私钥被密封住的纸钱包， 其他设计有密钥和地址的额外副本，类似于票根形式的可以拆卸存根，让你可以存储多个副本以防火灾、洪水或其他自然灾害。



图4-12 在备份“存根”上有多个私钥副本的纸钱包



“钱包”一词在比特币中有多重含义。 广义上，钱包是一个应用程序，为用户提供交互界面。钱包控制用户访问权限，管理密钥和地址，跟踪余额以及创建和签名交易。 狭义上，即从程序员的角度来看，“钱包”是指用于存储和管理用户密钥的数据结构。 我们将深入介绍第二层含义，本章中钱包是私钥的容器，一般是通过结构化文件或简单数据库来实现。

5.1钱包技术概述

在本节中，我们总结了各种技术，它们为用户构建起友好，安全和灵活的比特币钱包。

一个常见误解是，比特币钱包里含有比特币。 事实上，钱包里只含有钥匙。“钱币”被记录在比特币网络的区块链中。 用户通过钱包中的密钥签名交易，从而来控制网络上的钱币。 在某种意义上，比特币钱包是密钥链。

提示比特币钱包只含有密钥，而不是钱币。 每个用户有一个包含多个密钥的钱包。 钱包只包含私钥/公钥对的密钥链（请参阅[私钥章节]）。 用户用密钥签名交易，从而证明他们拥有交易输出（他们的钱币）。 钱币以交易输出的形式存储在区块链中（通常记为vout或txout）。

有两种主要类型的钱包，区别在于它们包含的多个密钥是否相互关联。

第一种类型是非确定性钱包（nondeterministic wallet），其中每个密钥都是从随机数独立生成的。密钥彼此无关。这种钱包也被称为“Just a Bunch Of Keys（一堆密钥）”，简称JBOK钱包。

第二种类型是确定性钱包（deterministic wallet），其中所有的密钥都是从一个主密钥派生出来，这个主密钥即为种子（seed）。该类型钱包中所有密钥都相互关联，如果有原始种子，则可以再次生成全部密钥。确定性钱包中使用了许多不同的密钥推导方法。最常用的推导方法是使用树状结构，称为分级确定性钱包或HD钱包。

确定性钱包由种子衍生创造。为了便于使用，种子被编码为英文单词，也称为助记词。

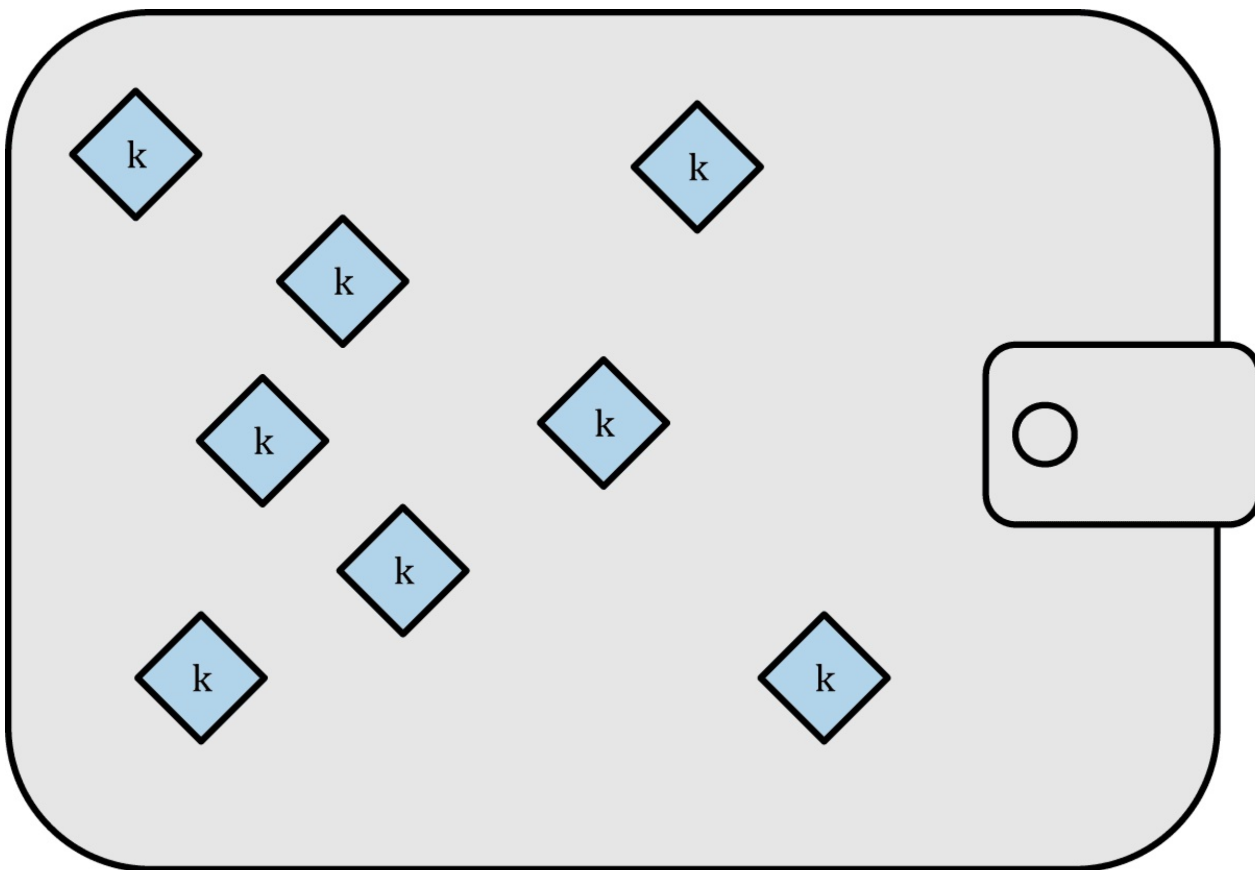
接下来的几节将深入介绍这些技术。

5.1.1非确定性（随机）钱包

在最早的一批比特币客户端中（ Bitcoin Core，现在称作比特币核心客户端），钱包只是随机生成的私钥集合。这种类型的钱包被称作零型非确定钱包。举个例子，比 特币核心客户端预先生成100个随机私钥，从最开始就生成足够多的私钥并且每个密钥只使用一次。这种钱包现在正在被确定性钱包替换，因为它们难以管理、 备份以及导入。随机密钥的缺点就是如果你生成很多私钥，你必须保存它们所有的副本。这就意味着这个钱包必须被经常性 地备份。每一个密钥都必须备份，否则一旦钱包不可访问时，钱包所控制的资金就付之东流。这种情况直接与避免地址重复使用的原则相冲突——每个比特币地址只能用一次交易。地址重复使用

将多个交易和地址关联在一起，这会减少隐私。当你想避免重复使用地址时，零型非确定性钱包并不是好的选择，因为你要创造过多的私钥并且要保存它们。虽然比特币核心客户端包含零型钱包，但比特币的核心开发者并不鼓励大家使用。

图5-1展示的是一个非确定性钱包，其含有的随机密钥是个松散的集合。

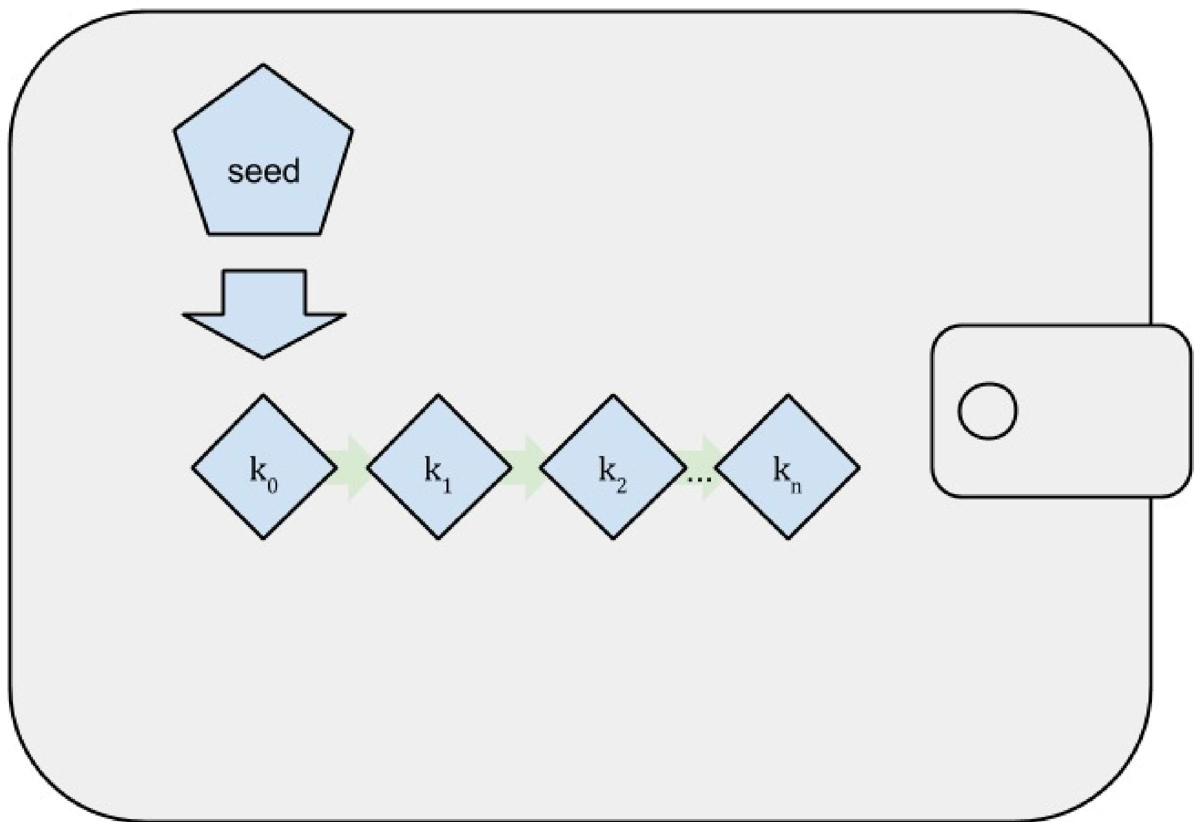


提示除了简单的测试之外，不要使用非确定性钱包。 它们对于备份和使用来说太麻烦了。 相反，推荐使用基于行业标准的HD钱包，可以用种子助记词进行备份。

5.1.2 确定性（种子）钱包

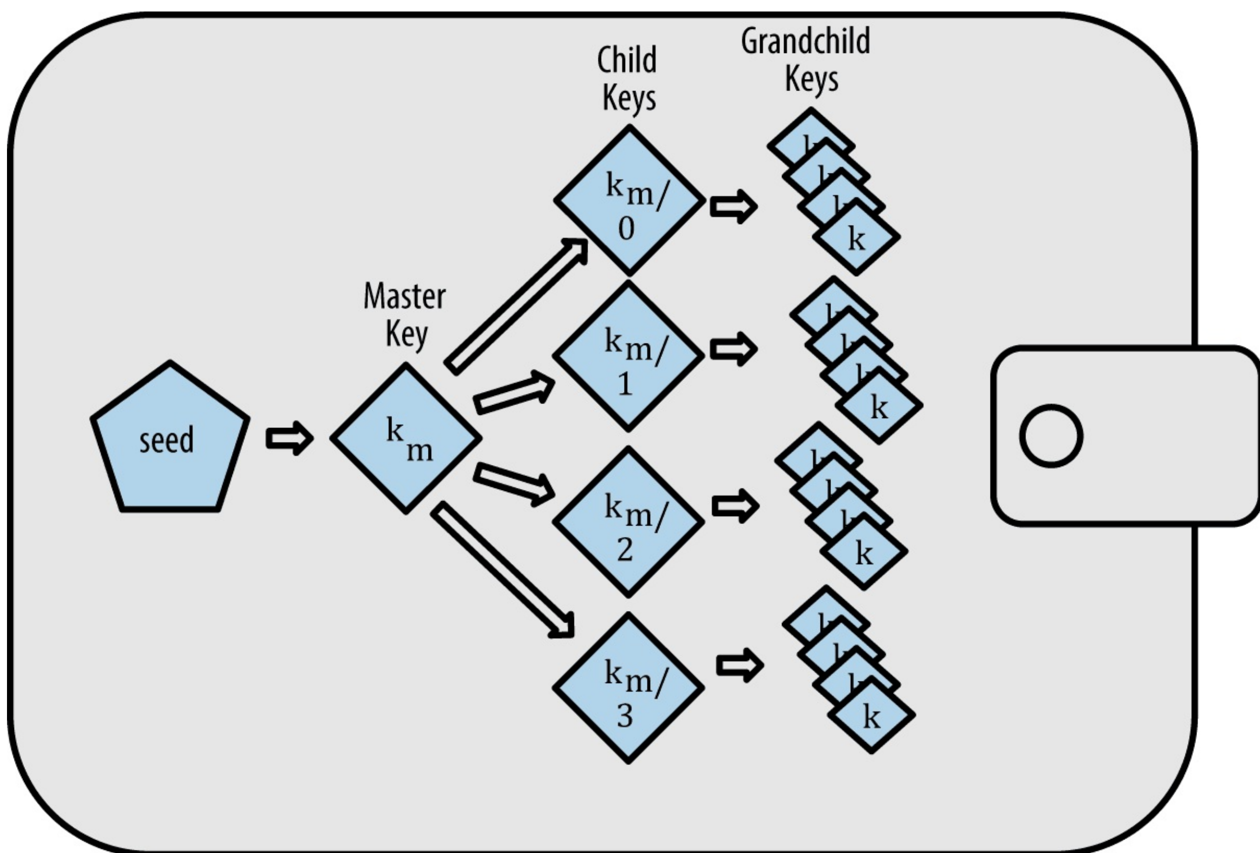
确定性，或者“种子”钱包包含通过使用单项离散函数而可从公共的种子生成的私钥。种子是随机生成的数字。这个数字也含有比如索引号码或者可生成私钥的“链码”（参见“ 分层确定性钱包（BIP0032/BIP0044）”一节）。在确定性钱包中，种子足够恢复所有的已经产生的私钥，所以只用在初始创建时的一个简单备份就足以搞定。并且种子也足够让钱包导入或者导出。这就很容易允许使用者的私钥在钱包之间轻松转移。

图5-2展示了确定性钱包的逻辑图。



5.1.3 分层确定性钱包（HD Wallets (BIP-32/BIP-44)）

确定性钱包被开发成更容易从单个“种子”中生成许多密钥。确定性钱包的最高级形式是通过BIP0032标准定义的HD钱包。HD钱包包含以树状结构衍生的密钥，使得父密钥可以衍生一系列子密钥，每个子密钥又可以衍生出一系列孙密钥，以此类推，无限衍生。图5-3展示了树状结构。



相比较随机（不确定性）密钥，HD钱包有两个主要的优势。第一，树状结构可以被用来表达额外的组织含义。比如当一个特定分支的子密钥被用来接收交易收入并且有另一个分支的子密钥用来负责支付花费。不同分支的密钥都可以被用在企业环境中，这就可以支配不同的分支部门、子公司、具体功能以及会计类别。

HD钱包的第二个好处就是它可以允许使用者去建立一个公共密钥的序列而不需要访问相对应的私钥。这可允许HD钱包在不安全的服务器中使用或者在每笔交易中发行不同的公共钥匙。公共钥匙不需要被预先加载或者提前衍生，而在服务器中不需要可用来支付的私钥。

5.1.4种子和助记词（BIP-39）

HD钱包具有管理多个密钥和地址的强大机制。由一系列英文单词生成种子是个标准化的方法，这样易于在钱包中转移、导出和导入，如果HD钱包与这种方法相结合，将会更加有用。这些英文单词被称为助记词，标准由BIP-39定义。 今天，大多数比特币钱包（以及其他加密货币的钱包）使用此标准，并可以使用可互操作的助记词导入和导出种子进行备份和恢复。

让我们从实际的角度来看以下哪种种子更容易抄录、阅读、导出以及导入。

16进制表示的种子： 0C1E24E5917779D297E14D45F14E1A1A

助记词表示的种子：

army van defense carry jealous true garbage claim echo media make crunch

5.1.5钱包最佳实践

由于比特币钱包技术已经成熟，出现了一些常见的行业标准，使得比特币钱包具备广泛互操作，易于使用，安全和灵活的特性。这些常用的标准是：

助记码，基于BIP-39

HD钱包，基于BIP-32

多用途HD钱包结构，基于BIP-43

多币种和多帐户钱包，基于BIP-44

这些标准可能会随着发展而改变或过时，但是现在它们形成了一套互锁技术，这些技术已成为比特币的事实上的钱包标准。

这些标准已被广泛的软件和硬件比特币钱包采用，使所有这些钱包互操作。用户可以导出在其中一个钱包上生成的助记符，并将其导入另一个钱包，实现恢复所有交易，密钥和地址。

列举支持这些标准的软件钱包，包括（按字母顺序排列）Breadwallet，Copay，Multibit HD和Mycelium。列举支持这些标准的硬件钱包，包括（按字母顺序排列）Keepkey，Ledger和Trezor。

以下部分将详细介绍这些技术。

提示如果您正准备开发一个比特币钱包，那么它应该被构建为一个HD钱包，一个种子被编码为助记词代码进行备份，遵循BIP-32，BIP-39，BIP-43和BIP-44标准，下面章节有所涉猎。

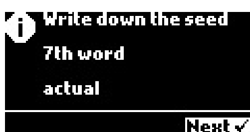
5.1.6使用比特币钱包

在[用户故事]中，我们介绍了Gabriel，里约热内卢是一个有进取心的少年，他正在经营一家简单的网络商店，销售比特币品牌的T恤，咖啡杯和贴纸。

Gabriel使用Trezor比特币硬件钱包（Trezor设备：硬件HD钱包）来安全地管理他的比特币。Trezor是一个简单的USB设备，具有两个按钮，用于存储密钥（以*HD*钱包的形式）和签署交易。 *Trezor*钱包遵循本章讨论的所有行业标准，因此Gabriel不依赖于任何专有技术或单一供应商解决方案。



当Gabriel首次使用Trezor时，设备从内置的硬件随机数生成器生成助记词和种子。 在这个初始化阶段，钱包在屏幕上按顺序逐个显示单词。



通过写下这个助记符，Gabriel创建了一个备份（参见表5-1），可以在Trezor设备丢失或损坏的情况下用于恢复。 在新的Trezor钱包，或者任一种兼容的软件和硬件钱包中，助记词都可以用于恢复。 请注意，单词序列很重要，因此，记忆纸备份需要对每个单词都有空格。 Gabriel必须仔细记录每个单词的编号，以保持正确的顺序。 表5-1Gabriel的助记器备份

1.	*army*	7.	*garbage*
2.	*van*	8.	*claim*
3.	*defense*	9.	*echo*
4.	*carry*	10.	*media*
5.	*jealous*	11.	*make*
6.	*true*	12.	*crunch*

提示为了简单起见，Gabriel的助记词记录中显示了一个12个词。 事实上，大多数硬件钱包生成更安全的24个词的助记符。 助记词以完全相同的方式使用，不管长度如何。

作为网店的第一次实践，Gabriel使用他的Trezor设备生成一个比特币地址。 所有客户的订单都使用此单一地址。 我们将看到，这种方法有一些缺点，不过可以使用HD钱包进行改进。

5.2钱包技术细节

现在我们来深入了解被众多比特币钱包所使用的重要的行业标准。

5.2.1助记码词汇（BIP-39）

助记码词汇是英文单词序列代表（编码）用作种子对应所确定性钱包的随机数。单词的序列足以重新创建种子，并且从 种子那里重新创造钱包以及所有私钥。在首次创建钱包时，带有助记码的，运行确定性钱包的钱包的应用程序将会向使用者展示一个12至24个词的顺序。单词的顺序就是钱包的备份。它也可以被用来恢复以及重新创造应用程序相同或者兼容的钱包的密钥。助记码词汇可以让使用者复制钱包更容易一些，因为相比较随机数字顺序来说，它们更容易地被阅读和正确抄写。

提示助记词经常与“脑钱包”混淆。 他们不一样。主要区别在于脑钱包由用户选择的单词组成，而助记符是由钱包随机创建的，并呈现给用户。 这个重要的区别使助记词更加安全，因为人类猜测随机数还是无能为力。

助记码被定义在比特币的改进建议39中（参见"附录２ 比特币改进协议[bip0039]”）。需要注意的是，BIP-39是助记码标准的一个实施方案。还有一个不同的标准，使用一组不同的单词，是由Electrum钱包使用，并且早于BIP-39。 BIP-39由Trezor硬件钱包背后的公司提出，与Electrum的实施不兼容。 然而，BIP-39现在已经在数十个可互操作的实践案例中获得了广泛的行业支持，应被视为事实上的行业标准。

BIP-39定义了助记符码和种子的创建，我们在这里描述了九个步骤。 为了清楚起见，该过程分为两部分：

1-6步是创建助记词，7-9步是从助记词到种子。

5.2.2创建助记词

助记词是由钱包使用BIP-39中定义的标准化过程自动生成的。 钱包从熵源开始，增加校验和，然后将熵映射到单词列表：

1、创建一个128到256位的随机序列（熵）。

2、提出SHA256哈希前几位（熵长/ 32），就可以创造一个随机序列的校验和。

3、将校验和添加到随机序列的末尾。

4、将序列划分为包含11位的不同部分。

5、将每个包含11位部分的值与一个已经预先定义2048个单词的字典做对应。

6、生成的有顺序的单词组就是助记码。

图5-6展示了熵如何生成助记词。

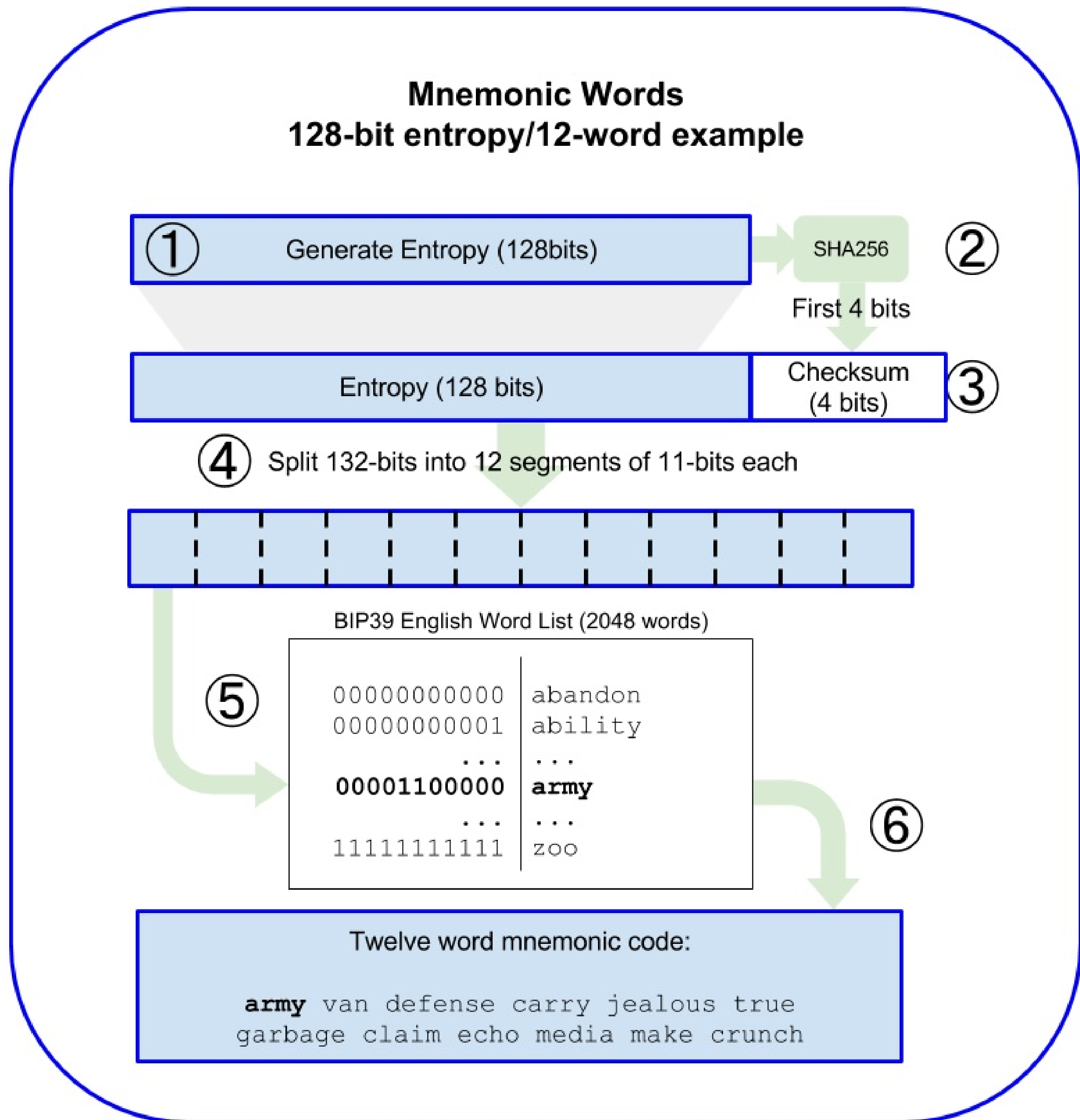


表5-2表示了熵数据的大小和助记词的长度之间的关系。

Entropy (bits)	Checksum (bits)	Entropy + checksum (bits)	Mnemonic length (words)
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21
256	8	264	24

5.2.3从助记词生成种子

助记词表示长度为128至256位的熵。 通过使用密钥延伸函数PBKDF2，熵被用于导出较长的（512位）种子。将所得的种子用于构建确定性钱包并得到其密钥。

密钥延伸函数有两个参数：助记词和盐。其中盐的目的是增加构建能够进行暴力攻击的查找表的困难度。 在BIP-39标准中，盐具有另一目的，它允许引入密码短语（passphrase），作为保护种子的附加安全因素，我们将在BIP-39可选密码短语章节详细地描述。

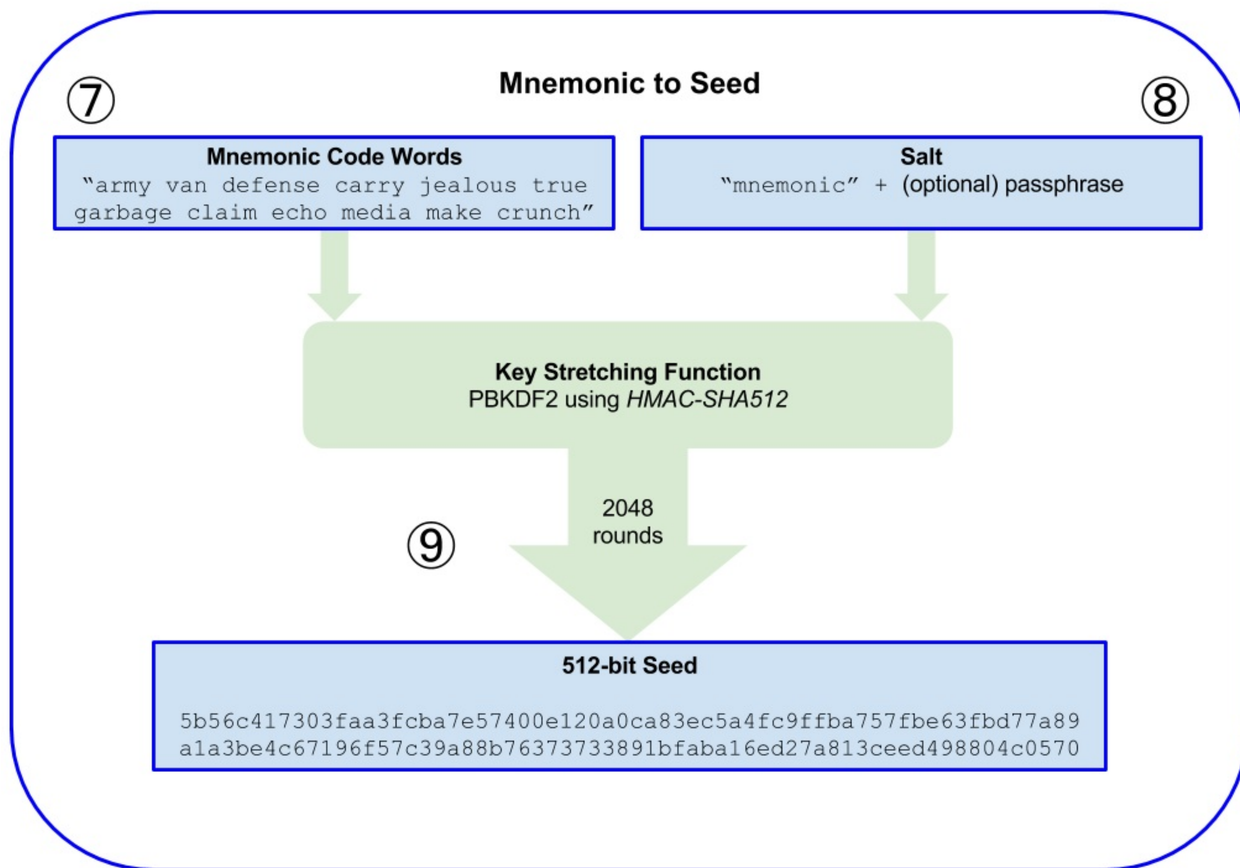
创建助记词之后的7-9步是：

7、PBKDF2密钥延伸函数的第一个参数是从步骤6生成的助记符。

8、PBKDF2密钥延伸函数的第二个参数是盐。 由字符串常数“助记词”与可选的用户提供的密码字符串连接组成。

9、PBKDF2使用HMAC-SHA512算法，使用2048次哈希来延伸助记符和盐参数，产生一个512位的值作为其最终输出。 这个512位的值就是种子。

图5-7显示了从助记词如何生成种子



提示 密钥延伸函数，使用2048次哈希是一种非常有效的保护，可以防止对助记词或密码短语的暴力攻击。 它使得攻击尝试非常昂贵（从计算的角度），需要尝试超过几千个密码和助记符组合，而这样可能产生的种子的数量是巨大的（2^512）。

表5-3、5-4和表5-5展示了一些助记码的例子和它所生成的种子。

Entropy input (128 bits)	0c1e24e5917779d297e14d45f14e1a1a
Mnemonic (12 words)	army van defense carry jealous true garbage claim echo media make crunch
Passphrase	(none)
Seed (512 bits)	5b56c417303faa3fcba7e57400e120a0ca83ec5a4fc9ffba757fbe63fbd77a89a1a3be4c67196f57c39 a88b76373733891bfaba16ed27a813ceed498804c0570

Entropy input (128 bits)	0c1e24e5917779d297e14d45f14e1a1a
Mnemonic (12 words)	army van defense carry jealous true garbage claim echo media make crunch
Passphrase	SuperDuperSecret
Seed (512 bits)	3b5df16df2157104cfdd22830162a5e170c0161653e3afe6c88defeefb0818c793dbb28ab3ab091897d0715861dc8a18358f80b79d49acf64142ae57037d1d54

Entropy input (256 bits)	2041546864449caff939d32d574753fe684d3c947c3346713dd8423e74abcf8c
Mnemonic (24 words)	cake apple borrow silk endorse fitness top denial coil riot stay wolf luggage oxygen faint major edit measure invite love trap field dilemma oblige
Passphrase	(none)
Seed (512 bits)	3269bce2674acbd188d4f120072b13b088a0ecf87c6e4cae41657a0bb78f5315b33b3a04356e53d062e55f1e0deaa082df8d487381379df848a6ad7e98798404

5.2.4BIP-39中的可选密码短语

BIP-39标准允许在推导种子时使用可选的密码短语。 如果没有使用密码短语，助记词是用由常量字符串“助记词”构成的盐进行延伸，从任何给定的助记词产生一个特定的512位种子。 如果使用密码短语，密钥延伸函数使用同样的助记词也会产生不同的种子。事实上，给予一个单一的助记词，每一个可能的密码短语都会导致不同的种子。 基本上没有“错误”的密码短语， 所有密码短语都是有效的，它们都会导致不同的种子，形成一大批可能未初始化的钱包。这批钱包非常之大（2^512），使用暴力破解或随机猜测基本不可能。

提示BIP-39中没有“错误的”密码短语。 每个密码都会导致一些钱包，只是未使用的钱包是空的。

可选密码短语带来两个重要功能：

（存储在大脑中的）密码短语成为第二个因素，使得助记词不能单独使用，避免了助记词备份盗取后被利用。 起到掩人耳目的效果，把密码短语指向有小额资金的钱包，分散攻击者注意力，使其不在关注拥有大额资金的“真实”钱包。

然而，需要注意的是，使用密码短语也会引起丢失的风险：

如果钱包所有者无行为能力或死亡，没有人知道密码，种子是无用的，所有存储在钱包中的资金都将永远丢失。相反，如果所有者将密码短语与种子备份在相同的地方，则违反了上述第二个因素的目的。虽然密码是非常有用的，但它们只能与仔细计划的备份和恢复流程结合使用，考虑到所有者个人风险的可能性，应该允许其家人恢复加密资产。

5.2.5使用助记符代码

BIP-39被做成函数库，支持多种编程语言：

python-mnemonic

SatoshiLabs团队在Python中提出了BIP-39标准的参考实现

bitcoinjs/bip39

作为流行的bitcoinJS框架的一部分，在JavaScript中实现了BIP-39

libbitcoin/mnemonic

作为流行的Libbitcoin框架的一部分，在C ++中实现了BIP-39

还有一个BIP-39生成器在独立的网页中实现，对于测试和实验非常有用。图5-8展示一个独立的网页，可以生成助记词、种子和扩展私钥。

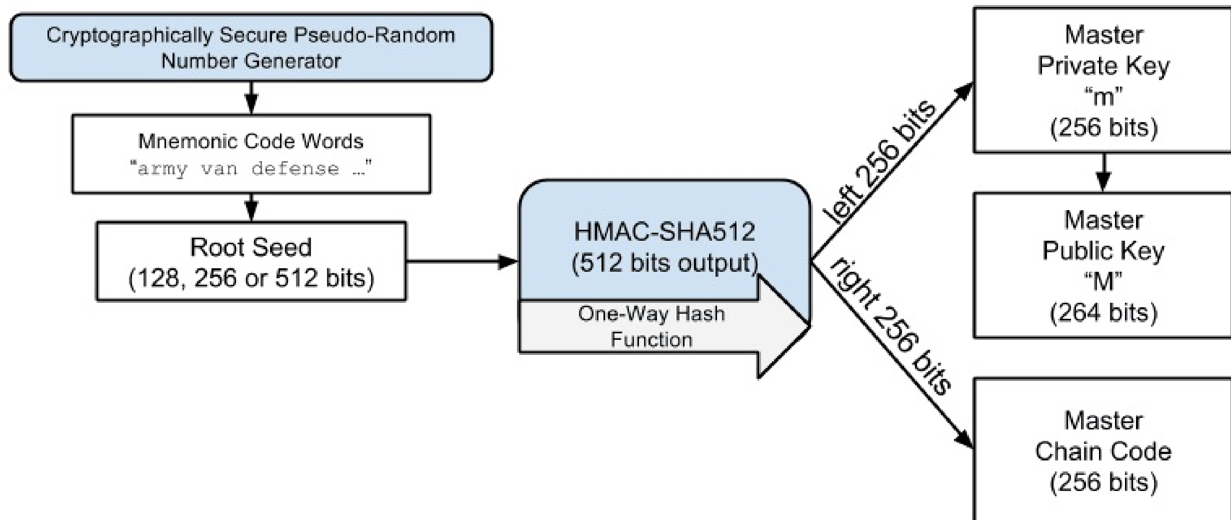
BIP-39生成器可以离线使用，也可以使用这个在线地址.地址转向到这儿了。

5.3从种子中创造HD钱包

HD钱包从单个根种子（root seed）中创建，为128到256位的随机数。最常见的是，这个种子是从助记符产生的，如上一节所述。

HD钱包的所有的确定性都衍生自这个根种子。任何兼容HD 钱包的根种子也可重新创造整个HD钱包。所以简单的转移HD钱包的根种子就让HD钱包中所包含的成千上百万的密钥被复制，储存导出以及导入。

图5-9展示创建主密钥以及HD钱包的主链代码的过程。



根种子输入到HMAC-SHA512算法中就可以得到一个可用来创造主私钥(m)（master private key(m)）和主链代码（a master chain code）的哈希。主私钥（m）之后可以通过使用我们在本章先前看到的那个普通椭圆曲线m * G过程生来成相对应的主公钥（M）。 链代码用于从母密钥中创造子密钥的那个函数中引入熵。如下一节所示。

5.3.1私有子密钥的衍生

分层确定性钱包使用CKD（child key derivation)函数去从母密钥衍生出子密钥。

子密钥衍生函数是基于单项哈希函数。这个函数结合了：

一个母私钥或者公共钥匙（ECDSA未压缩键）

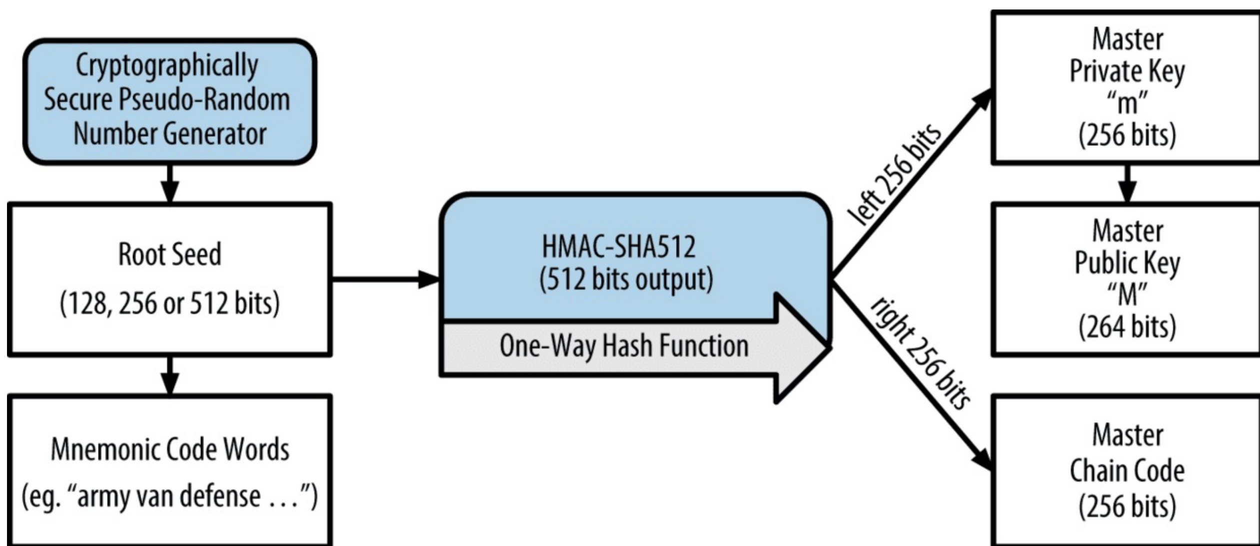
一个叫做链码（256 bits）的种子

一个索引号（32 bits）

链码是用来给这个过程引入确定性随机数据的，使得索引不能充分衍生其他的子密钥。因此，有了子密钥并不能让它发现自己的姊妹密钥，除非你已经有了链码。最初的链码种子（在密码树的根部）是用随机数据构成的，随后链码从各自的母链码中衍生出来。

这三个项目（母密钥，链码，索引）相结合并散列可以生成子密钥，如下。

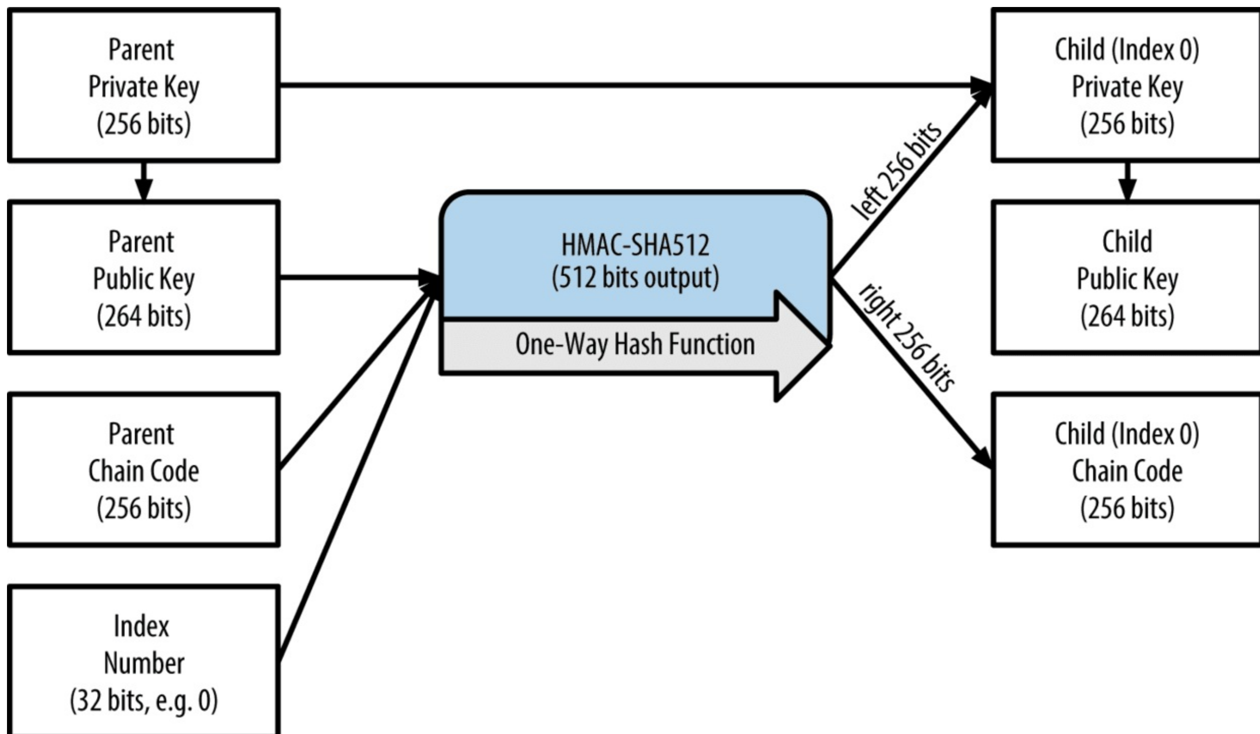
母公共钥匙——链码——以及索引号合并在一起并且用HMAC-SHA512函数散列之后可以产生512位的散列。所得的散列可被拆分为两部分。散列右半部分的256位产出可以给子链当链码。左半部分256位散列以及索引码被加载在母私钥上来衍生子私钥。在图5-11中，我们看到这个说明——索引集被设为0去生产母密钥的第0个子密钥（第一个通过索 引）。



改变索引可以让我们延长母密钥以及创造序列中的其他子密钥。比如子0，子1，子2等等。每一个母密钥可以有2,147,483,647 (2^31) 个子密钥。2^31是整个2^32范围可用的一半，因为另一半是为特定类型的推导而保留的，我们将在本章稍后讨论。

向密码树下一层重复这个过程，每个子密钥可以依次成为母密钥继续创造它自己的子密钥，直到无限代。

图5-11阐述了扩展母公钥来衍生子公钥的传递机制。



5.3.2使用衍生的子密钥

子私钥不能从非确定性（随机）密钥中被区分出来。因为衍生函数是单向的，所以子密钥不能被用来发现他们的母密钥。子密钥也不能用来发现他们的相同层级的姊妹密钥。如果你有第n个子密钥，你不能发现它前面的（第n－1）或者 后面的子密钥（n+1）或者在同一顺序

中的其他子密钥。只有母密钥以及链码才能得到所有的子密钥。没有子链码的话，子密钥也不能用来衍生出任何孙密钥。你需要同时有子密钥以及对应的链码才能创建一个新的分支来衍生出孙密钥。

那子私钥自己可被用做什么呢？它可以用来做公钥和比特币地址。之后它就可以被用在那个地址来签署交易和支付任何东西。

提示子私钥、对应的公钥以及比特币地址都不能从随机创造的密钥和地址中被区分出来。事实是它们所在的序列，在创造他们的HD钱包函数之外是不可见的。一旦被创造出来，它们就和“正常”密钥一样运行了。

5.3.3扩展密钥

正如我们之前看到的，密钥衍生函数可以被用来创造密钥树上任何层级的子密钥。这只需要三个输入量：一个密钥，一个链码以及想要的子密钥的索引。密钥以及链码这两个重要的部分被结合之后，就叫做扩展密钥（extended key）。术语“extended key”也被认为是“可扩展的密钥”，因为这种密钥可以用来衍生子密钥。

扩展密钥可以简单地被储存并且表示为简单的将256位密钥与256位链码所并联的512位序列。有两种扩展密钥。扩展的私钥是私钥以及链码的结合。它可被用来衍生子私钥（子私钥可以衍生子公钥）。公钥以及链码组成扩展公钥，它可以用来扩展子公钥，见“生成公钥”章节。

想象一个扩展密钥作为HD钱包中密钥树结构的一个分支的根。你可以衍生出这个分支的剩下所有部分。扩展私钥可以创建一个完整的分支，而扩展公钥只能够创造一个公钥的分支。

提示一个扩展密钥包括一个私钥（或者公钥）以及一个链码。一个扩展密钥可以创造出子密钥并且能创造出密钥树结构中的整个分支。分享扩展密钥就可以访问整个分支。

扩展密钥通过Base58Check来编码，从而能轻易地在不同的BIP-32兼容钱包之间导入导出。扩展密钥编码用的 Base58Check使用特殊的版本号，这导致在Base58编码字符中，出现前缀“xprv”和“xpub”。这种前缀可以让编码更易被识别。因为扩展密钥是512或者513位，所以它比我们之前所看到的Base58Check编码串更长一些。

以下面的扩展私钥为例，其使用的是Base58Check编码：

xprv9tyUQV64JT5qs3RSTJkXCWKMyUgoQp7F3hA1xzG6ZGu6u6Q9VMNjGr67Lctvy5P8oyaYAL9CAWrUE9i6GoNMKUga5biW6Hx4tws2six3b9c

这是上面扩展私钥对应的扩展公钥，同样使用Base58Check编码：

xpub67xpozcx8pe95XVuZLHXZeG6XWXHpGq6Qv5cmNfi7cS5mtjJ2tgypeQbBs2UAR6KECeeMVKZBPLrtJunSDMstweyLXhRgPxdp14sk9tJPW9

5.3.4公共子密钥推导

正如之前提到的，分层确定性钱包的一个很有用的特点就是可以不通过私钥而直接从公共母密钥派生出公共子密钥的能 力。这就给了我们两种衍生子公钥的方法：或者通过子私钥，再或者就是直接通过母公钥。

因此，扩展密钥可以在HD钱包结构的分支中，被用来衍生所有的公钥（且只有公钥）。

这种快捷方式可以用来创造非常保密的只有公钥配置。在配置中，服务器或者应用程序不管有没有私钥，都可以有扩展公钥的副本。这种配置可以创造出无限数量的公钥以及比特币地址。但是发送到这个地址里的任何比特币都不能使用。与此同时，在另一种更保险的服务器上，扩展私钥可以衍生出所有的对应的可签署交易以及花钱的私钥。

这种方案的常见应用是安装扩展公钥电商的网络服务器上。网络服务器可以使用这个公钥衍生函数去给每一笔交易（比如客户的购物车）创造一个新的比特币地址。但为了避免被偷，网络服务商不会有任何私钥。没有HD钱包的话，唯一的方法就是在不同的安全服务器上创造成千上万个比特币地址，之后就提前上传到电商服务器上。这种方法比较繁琐而且要求持续的维护来确保电商服务器不“用光”公钥。

这种解决方案的另一种常见的应用是冷藏或者硬件钱包。在这种情况下，扩展的私钥可以被储存在纸质钱包中或者硬件设备中（比如 Trezor 硬件钱包），与此同时扩展公钥可以在线保存。使用者可以根据意愿创造“接收”地址而私钥可以安全地在线下被保存。为了支付资金，使用者可以使用扩展的私钥离线签署比特币客户或者通过硬件钱包设备（比如 Trezor）签署交易。

5.3.5在网店中使用扩展公钥（xpub）

继续Gabriel网店的故事，让我们看看Gabriel是如何使用HD钱包。

Gabriel在一个网络上的托管服务器上建立一个简单的WordPress页面，作为他的网上商店。它的网店非常简单，只有几个页面和一张带有一个比特币地址的订单。

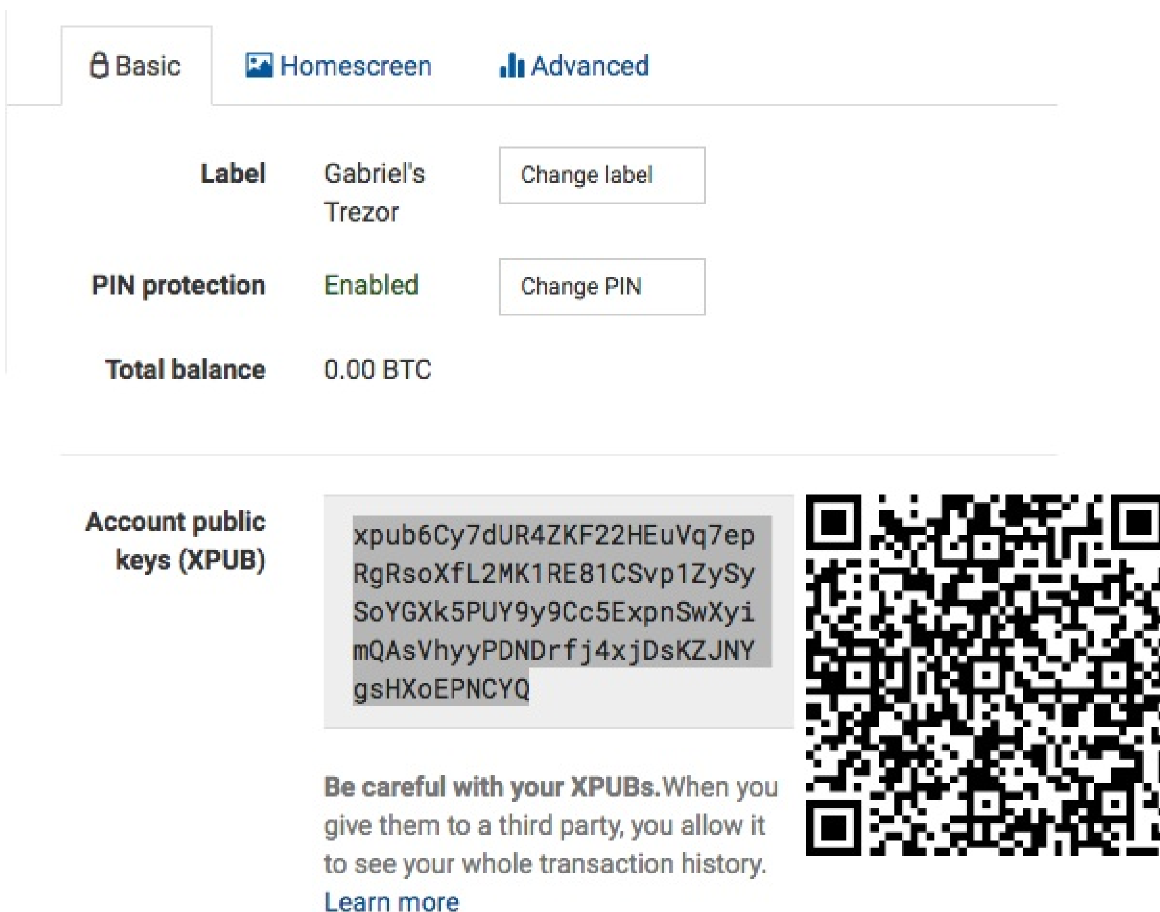
Gabriel使用他的Trezor设备生成的第一个比特币地址作为他的商店的主要比特币地址。这样，所有收到的付款都将支付给他的Trezor硬件钱包所控制的地址。

客户可以使用表格提交订单，并向Gabriel发布的比特币地址发送付款，触发一封电子邮件，其中包含Gabriel的订单详细信息。每周只几个订单，这个系统运行得很好。

然而，这个小型网络商店变得相当成功，并吸引了很多来自当地社区的订单。Gabriel很快就不堪重负。由于所有订单都支付相同的地址，因此很难正确匹配订单和交易，特别是当同一数量的多个订单紧密相连时。

HD钱包可以在不知道私钥的情况下获取公共子密钥，该能力为Gabriel提供了更好的解决方案。 Gabriel可以在他的网站上加载一个扩展公钥（xpub），这可以用于为每个客户订单导出唯一的地址。Gabriel可以花费他在Trezor里资金，但加载在网站上的xpub只能生成地址并收到资金。HD钱包的这个功能非常安全。 Gabriel的网站不包含任何私钥，因此不需要高级别的安全性。

为了导出xpub，Gabriel将基于Web的软件与Trezor硬件钱包配合使用。必须插入Trezor设备才能导出公钥。请注意，硬件钱包永远不会导出私钥，这些密钥始终保留在设备上。图5-12显示了Gabriel用于导出xpub的Web界面。

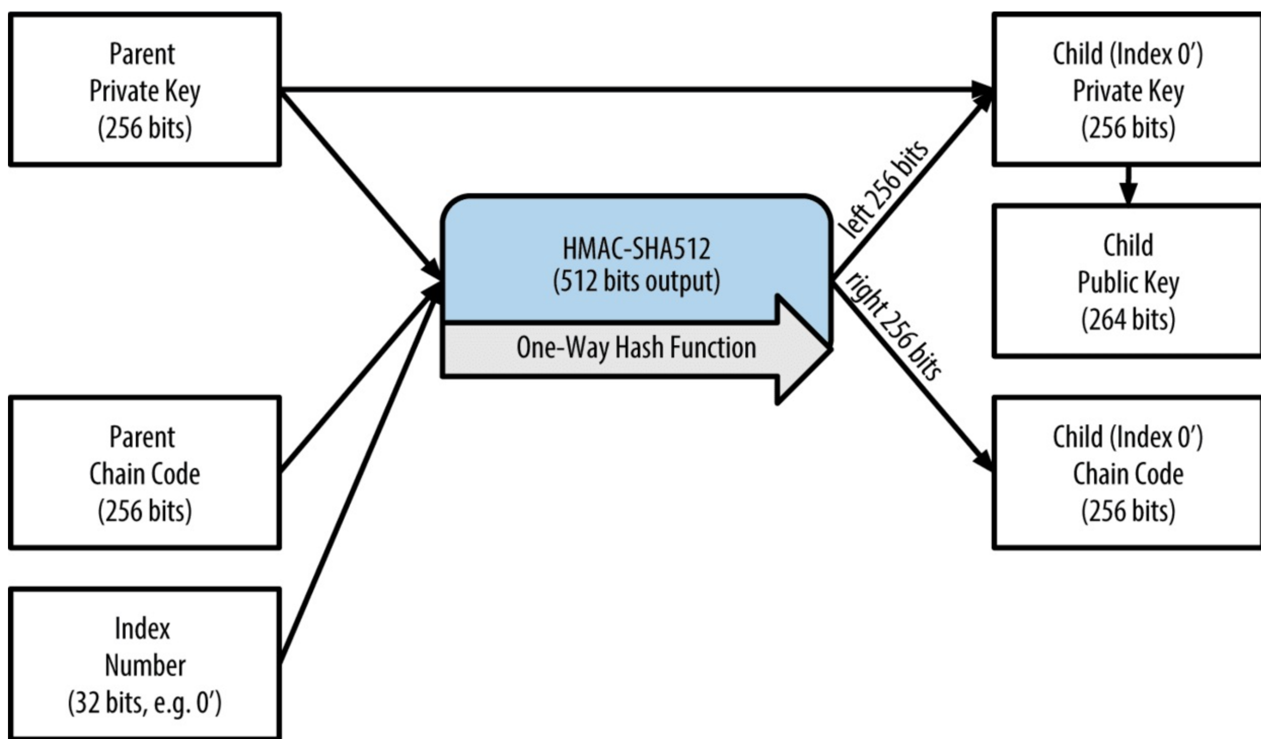


Gabriel将xpub复制到他网店的比特币购物软件中。 他使用的软件是Mycelium Gear，它是一个网店的开源插件，用于各种网络托管和内容平台。 Mycelium Gear使用xpub为每次购买生成一个唯一的地址。

5.3.6硬化子密钥的衍生

从扩展公钥衍生一个分支公钥的能力是很重要的，但牵扯一些风险。访问扩展公钥并不能得到访问子私钥的途径。但是，因为扩展公钥包含有链码，如果子私钥被知道或者被泄漏的话，链码就可以被用来衍生所有的其他子私钥。简单地泄露的私钥以及一个母链码，可以暴露所有的子密钥。更糟糕的是，子私钥与母链码可以用来推断母私钥。

为了应对这种风险，HD钱包使用一种叫做硬化衍生(hardened derivation）的替代衍生函数。这就“打破”了母公钥以及子链码之间的关系。这个硬化衍生函数使用了母私钥去推导子链码，而不是母公钥。这就在母/子顺序中创造了一道“防火墙”——有链码但并不能够用来推算子链码或者姊妹私钥。强化衍生函数看起来几乎与一般的衍生的子私钥相同，不同的是母私钥被用来输入散列函数中而不是母公钥，如图5-13所示。



当强化私钥衍生函数被使用时，得到的子私钥以及链码与使用一般衍生函数所得到的结果完全不同。得到的密钥"分支"可以被用来生产不易被攻击的扩展公钥，因为它所含的链码不能被用来开发或者暴露任何私钥。强化衍生也因此被用在上一层级，使用扩展公钥的密钥树中创造"间隙"。

简单地来说，如果你想要利用扩展公钥的便捷来衍生公钥的分支而不将你自己暴露在泄露扩展链码的风险下， 你应该从强化母私钥衍生公钥，而不是从一般的母私钥来衍生。最好的方式是，为了避免了推到出主密钥，主密钥所衍生的第一层级的子密钥最好使用强化衍生。

5.3.7正常衍生和强化衍生的索引号码

用在衍生函数中的索引号码是32位的整数。为了区分密钥是从正常衍生函数中衍生出来还是从强化衍生函数中产出，这个索引号被分为两个范围。索引号在0和2^31–1(0x0 to 0x7FFFFFFF)之间的是只被用在常规衍生。索引号在2^31和2^32– 1(0x80000000 to 0xFFFFFFFF)之间的只被用在强化衍生。因此，索引号小于2^31就意味着子密钥是常规的，而大于或者等于2^31的子密钥就是强化型的。

为了让索引号码更容易被阅读和展示，强化子密钥的索引号码是从0开始展示的，但是右上角有一个小撇号。第一个常规子密钥因此被表述为0，但是第一个强化子密钥（索引号为0x80000000）就被表示为0'。第二个强化密钥依序有了索 引号0x80000001，且被显示为1'，以此类推。当你看到HD钱包索引号i'，这就意味着 2^31+i。

5.3.8HD钱包密钥识别符（路径）

HD钱包中的密钥是用“路径”命名的，且每个级别之间用斜杠（/）字符来表示（见表5-6）。由主私钥衍生出的私钥起始以“m”打头。由主公钥衍生的公钥起始以“M“打头。因此，母密钥生成的第一个子私钥是m/0。第一个公钥是M/0。第一个子密钥的子密钥就是m/0/1，以此类推。

密钥的“祖先”是从右向左读，直到你达到了衍生出的它的主密钥。举个例子，标识符m/x/y/z描述的是子密钥m/x/y的第z个子密钥。而子密钥m/x/y又是m/x的第y个子密钥。m/x又是m的第x个子密钥。

HD path	Key described
m/0	The first (0) child private key from the master private key (m)
m/0/0	The first grandchild private key of the first child (m/0)
m/0'/0	The first normal grandchild of the first *hardened* child (m/0')
m/1/0	The first grandchild private key of the second child (m/1)
M/23/17/0/0	The first great-great-grandchild public key of the first great-grandchild of the 18th grandchild of the 24th child

5.3.9HD钱包树状结构的导航

HD钱包树状结构提供了极大的灵活性。每一个母扩展密钥有40亿个子密钥：20亿个常规子密钥和20亿个强化子密钥。 而每个子密钥又会有40亿个子密钥并且以此类推。只要你愿意，这个树结构可以无限类推到无穷代。但是，又由于有了这个灵活性，对无限的树状结构进行导航就变得异常困难。尤其是对于在不同的HD钱包之间进行转移交易，因为内部组织到内部分支以及亚分支的可能性是无穷的。

两个比特币改进建议（BIPs）提供了这个复杂问题的解决办法——通过创建几个HD钱包树的提议标准。BIP-43提出使用第一个强化子索引作为特殊的标识符表示树状结构的“purpose”。基于BIP-43，HD钱包应该使用且只用第一层级的树的分支，而且有索引号码去识别结构并且有命名空间来定义剩余的树的目的地。举个例子，HD钱包只使用分支m/i'/是 为了表明那个被索引号“i”定义的特殊为目地。

在BIP-43标准下，为了延长的那个特殊规范，BIP-44提议了多账户结构作为“purpose”。所有遵循BIP-44的HD钱包依据只使用树的第一个分支的要求而被定义：m/44'/。 BIP-44指定了包含5个预定义树状层级的结构：

m / purpose' / coin_type' / account' / change / address_index

第一层的purpose总是被设定为44'。

第二层的“coin_type”特指币种并且允许多元货币HD钱包中的货币在第二个层级下有自己的亚树状结构。目前有三种货币被定义：Bitcoin is m/44'/0'、Bitcoin Testnet is m/44'/1'，以及Litecoin is m/44'/2'。

树的第三层级是“account”，这可以允许使用者为了会计或者组织目的，而去再细分他们的钱包到独立的逻辑性亚账户。 举个例子，一个HD钱包可能包含两个比特币“账户”：m/44'/0'/0' 和 m/44'/0'/1'。每个账户都是它自己亚树的根。

第四层级就是“change”。每一个HD钱包有两个亚树，一个是用来接收地址一个是用来创造找零地址。注意无论先前的层级是否使用强化衍生，这一层级使用的都是常规衍生。这是为了允许这一层级的树可以在不安全环境下，输出扩展公钥。

被HD钱包衍生的可用的地址是第四层级的子级，就是第五层级的树的“address_index”。比如，第三个层级的主账户收到比特币支付的地址就是 M/44'/0'/0'/0/2。表5-7展示了更多的例子。

HD路径	主要描述
M/44'/0'/0'/0/2	第三个收到公共钥匙的主比特币账户
M/44'/0'/3'/1/14	第十五改变地址公钥的第四个比特币账户
m/44'/2'/0'/0/1	为了签署交易的在莱特币主账户的第二个私钥

6.1 简介

比特币交易是比特币系统中最重要的部分。根据比特币系统的设计原理，系统中任何其他的部分都是为了确保比特币交易可以被生成、能在比特币网络中得以传播和通过验证，并最终添加入全球比特币交易总账簿（比特币区块链）。比特币交易的本质是数据结构，这些数据结构中含有比特币交易参与者价值转移的相关信息。比特币区块链是一本全球复式记账总账簿，每个比特币交易都是在比特币区块链上的一个公开记录。

在这一章，我们将会剖析比特币交易的多种形式、所包含的信息、如何被创建、如何被验证以及如何成为所有比特币交易永久记录的一部分。当我们在本章中使用术语“钱包”时，我们指的是构建交易的软件，而不仅仅是密钥的数据库。

6.2交易细节

在[第二章比特币概述]中，我们使用区块浏览器查看了Alice曾经在Bob的咖啡店（Alice与Bob's Cafe的交易）支付咖啡的交易。

区块浏览器应用程序显示从Alice的“地址”到Bob的“地址”的交易。 这是一个非常简化的交易中包含的内容。 实际上，正如我们将在本章中看到的，所显示的大部分信息都是由区块浏览器构建的，实际上并不在交易中。

Transaction View information about a bitcoin transaction

0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2

1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK (0.1 BTC - Output)

1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA - (Unspent) 0.015 BTC
1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK - (Unspent) 0.0845 BTC

97 Confirmations 0.0995 BTC

Summary	
Size	258 (bytes)
Received Time	2013-12-27 23:03:05
Included In Blocks	277316 (2013-12-27 23:11:54 +9 minutes)

Inputs and Outputs	
Total Input	0.1 BTC
Total Output	0.0995 BTC
Fees	0.0005 BTC
Estimated BTC Transacted	0.015 BTC

图1. Alice与Bob's Cafe的交易

6.2.1交易 - 幕后细节

在幕后，实际的交易看起来与典型的区块浏览器提供的交易非常不同。 事实上，我们在各种比特币应用程序用户界面中看到的大多数高级结构实际上并不存在于比特币系统中。

我们可以使用Bitcoin Core的命令行界面（getrawtransaction和decodeawtransaction）来检索Alice的“原始”交易，对其进行解码，并查看它包含的内容。 结果如下： Alice的交易被解码后是这个样子：

```
{
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid":"7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "scriptSig": "3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75
c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL] 0484
ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec345
7eee41c04f4938de5cc17b4a10fa336a8d752adf",
      "sequence": 4294967295
    }
 ],
  "vout": [
    {
      "value": 0.01500000,
      "scriptPubKey": "OP_DUP OP_HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 OP_E
QUALVERIFY OP_CHECKSIG"
    },
    {
      "value": 0.08450000,
      "scriptPubKey": "OP_DUP OP_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP_E
QUALVERIFY OP_CHECKSIG",
    }
  ]
}
```

您可能会注意到这笔交易似乎少了些什么东西，比如：Alice的地址在哪里？Bob的地址在哪里？ Alice发送的“0.1”个币的输入在哪里？ 在比特币里，没有具体的货币，没有发送者，没有接收者，没有余额，没有帐户，没有地址。为了使用者的便利，以及使事情更容易理解，所有这些都构建在更高层次上。

你可能还会注意到很多奇怪和难以辨认的字段以及十六进制字符串。 不必担心，本章将详细介绍这里所示的各个字段。

6.3交易的输入输出

比特币交易中的基础构建单元是交易输出。 交易输出是比特币不可分割的基本组合，记录在区块上，并被整个网络识别为有效。 比特币完整节点跟踪所有可找到的和可使用的输出，称为 “未花费的交易输出”（unspent transaction outputs），即UTXO。 所有UTXO的集合被称为UTXO集，目前有数百万个UTXO。 当新的UTXO被创建，UTXO集就会变大，当UTXO被消耗时，UTXO集会随着缩小。每一个交易都代表UTXO集的变化（状态转换）。

当我们说用户的钱包已经“收到”比特币时，我们的意思是，钱包已经检测到了可用的UTXO。通过钱包所控制的密钥，我们可以把这些UTXO花出去。 因此，用户的比特币“余额”是指用户钱包中可用的UTXO总和，而他们可能分散在数百个交易和区块中。 “一个用户的比特币余额”，这个概念是比特币钱包应用创建的派生之物。比特币钱包通过扫描区块链并聚集所有属于该用户的UTXO来计算该用户的余额 。大多数钱包维护一个数据库或使用数据库服务来存储所有UTXO的快速参考集，这些UTXO由用户所有的密钥来控制花费行为。

一个UTXO可以是1“聪”（satoshi）的任意倍数（整数倍）。就像美元可以被分割成表示两位小数的“分”一样，比特币可以被分割成八位小数的“聪”。尽管UTXO可以是任意值，但一旦被创造出来，即不可分割。这是UTXO值得被强调的一个重要特性：UTXO是面值为“聪”的离散（不连续）且不可分割的价值单元，一个UTXO只能在一次交易中作为一个整体被消耗。

如果一个 UTXO比一笔交易所需量大，它仍会被当作一个整体而消耗掉，但同时会在交易中生成零头。例如，你有 一个价值20比特币的 UTXO并且想支付1比特币，那么你的交易必须消耗掉整个20比特币的UTXO，并产生两个输出：一个支付了1比特币给接收人，另一个支付了19比特币的找零到你的钱包。这样的话，由于UTXO（或交易输出）的不可分割特性，大部分比特币交易都会产生找零。

想象一下，一位顾客要买1.5元的饮料。她掏出钱包并试图从所有硬币和钞票中找出一种组合来凑齐她要支付的1.5 元。如果可能的话，她会选刚刚好的零钱（比如一张1元纸币和5个一毛硬币）或者是小面额的组合（比如3个五毛硬币）。如果都不行的话，她会用一张大面额的钞票，比如5元纸币。如果她把5元给了商店老板，她会得到3.5元的找零，并把找零放回她的钱包以供未来的交易使用。

类似的，一笔比特币交易可以是任意金额，但必须从用户可用的UTXO中创建出来。用户不能再把UTXO进一步细分，就像不能把一元纸币撕开而继续当货币使用一样。用户的钱包应用通常会从用户可用的UTXO中选取多个来拼凑出一个大于或等于一笔交易所需的比特币量。

就像现实生活中一样，比特币应用可以使用一些策略来满足付款需求：组合若干小额UTXO，并算出准确的找零；或者使用一个比交易额大的UTXO然后进行找零。所有这些复杂的、由可花费UTXO组成的集合，都是由用户的钱包自动完成， 并不为用户所见。只有当你以编程方式用UTXO来构建原始交易时，这些才与你有关。

一笔交易会消耗先前的已被记录（存在）的UTXO，并创建新的UTXO以备未来的交易消耗。通过这种方式，一定数量的比特币价值在不同所有者之间转移，并在交易链中消耗和创建UTXO。一笔比特币交易通过使用所有者的签名来解锁UTXO，并通过使用新的所有者的比特币地址来锁定并创建UTXO。

从交易的输出与输入链角度来看，有一个例外，即存在一种被称为“币基交易”（Coinbase Transaction）的特殊交易，它是每个区块中的第一笔交易，这种交易存在的原因是作为对挖矿的奖励，创造出全新的可花费比特币用来支付给“赢家”矿工。这也就是为什么比特币可以在挖矿过程中被创造出来，我们将在“挖矿”这一章进行详述。

> 小贴士：输入和输出，哪一个是先产生的呢？先有鸡还是先有蛋呢？严格来讲，先产生输出，因为可以创造新比特币的 “币基交易”没有输入，但它可以无中生有地产生输出。

6.3.1 交易输出

每一笔比特币交易都会创造输出，并被比特币账簿记录下来。除特例之外（见“数据输出操作符”（OP_RETURN）），几乎所有的输出都能创造一定数量的可用于支付的比特币，也就是UTXO。这些UTXO被整个网络识别，所有者可在未来的交易中使用它们。

UTXO在UTXO集（UTXOset）中被每一个全节点比特币客户端追踪。 新的交易从UTXO集中消耗（花费）一个或多个输出。

交易输出包含两部分：

- 一定量的比特币，面值为“聪”（satoshis） ，是最小的比特币单位；
- 确定花费输出所需条件的加密难题（cryptographic puzzle）

这个加密难题也被称为锁定脚本(locking script), 见证脚本(witness script), 或脚本公钥(scriptPubKey)。

有关交易脚本语言会在后面121页的“交易脚本和脚本语言”一节中详细讨论。

现在，我们来看看 Alice 的交易（之前的章节“交易 - 幕后”所示），看看我们是否可以找到并识别输出。 在 JSON 编码中，输出位于名为 vout 的数组（列表）中：

```
"vout": [
  {
    "value": 0.01500000,
    "scriptPubKey": "OP_DUP OP_HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 OP_EQU
ALVERIFY
OP_CHECKSIG"
  },
  {
    "value": 0.08450000,
    "scriptPubKey": "OP_DUP OP_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP_EQU
ALVERIFY OP_CHECKSIG",
  }
]
```

如您所见，交易包含两个输出。 每个输出都由一个值和一个加密难题来定义。 在 Bitcoin Core 显示的编码中，该值显示以 bitcoin 为单位，但在交易本身中，它被记录为以 satoshis 为单位的整数。 每个输出的第二部分是设定支出条件的加密难题。 Bitcoin Core 将其显示为 scriptPubKey，并向我们展示了一个可读的脚本表示。

稍后将在脚本构造（Lock + Unlock）中讨论锁定和解锁UTXO的主题。 在 ScriptPubKey 中用于编辑脚本的脚本语言在章节Transaction Scripts（交易脚本）和Script Language（脚本语言）中讨论。 但在我们深入研究这些话题之前，我们需要了解交易输入和输出的整体结构。

6.3.1.1 交易序列化 - 输出

当交易通过网络传输或在应用程序之间交换时，它们被序列化。 序列化是将内部的数据结构表示转换为可以一次发送一个字节的格式（也称为字节流）的过程。 序列化最常用于编码通过网络传输或用于文件中存储的数据结构。 交易输出的序列化格式如下表所示：

Size	Field	Description
8 bytes (little-endian)	Amount	Bitcoin value in satoshis (10^{-8} bitcoin)
1–9 bytes (VarInt)	Locking-Script Size	Locking-Script length in bytes, to follow
Variable	Locking-Script	A script defining the conditions needed to spend the output

大多数比特币函数库和架构不会在内部将交易存储为字节流，因为每次需要访问单个字段时，都需要复杂的解析。为了方便和可读性，比特币函数库将交易内部存储在数据结构（通常是面向对象的结构）中。

从交易的字节流表示转换为函数库的内部数据结构表示的过程称为反序列化或交易解析。转换回字节流以通过网络传输、哈希化（hashing）或存储在磁盘上的过程称为序列化。大多数比特币函数库具有用于交易序列化和反序列化的内置函数。

看看是否可以从序列化的十六进制形式手动解码 Alice 的交易中，找到我们以前看到的一些元素。包含两个输出的部分在下面中已加粗显示：

0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd73
4d2804fe65fa35779000000008b483045022100884d142d86652a3f47
ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039
ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813
01410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade84
16ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc1
7b4a10fa336a8d752adffffffff02**60e31600000000001976a914ab6**
8025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef800000000000
1976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac 00000000

这里有一些提示：

- 加粗显示的部分有两个输出，每个都如本节之前所述进行了序列化。

- 0.015比特币的价值是1,500,000 satoshis。 这是十六进制的16 e3 60。
- 在串行化交易中，值16 e3 60以小端（最低有效字节优先）字节顺序进行编码，所以它看起来像60 e3 16。
- scriptPubKey的长度为25个字节，以十六进制显示为19个字节。

6.3.2交易输入

交易输入将UTXO（通过引用）标记为将被消费，并通过解锁脚本提供所有权证明。

要构建一个交易，一个钱包从它控制的UTXO中选择足够的价值来执行被请求的付款。 有时一个UTXO就足够，其他时候不止一个。 对于将用于进行此付款的每个UTXO，钱包将创建一个指向UTXO的输入，并使用解锁脚本解锁它。

让我们更详细地看一下输入的组件。输入的第一部分是一个指向UTXO的指针，通过指向UTXO被记录在区块链中所在的交易的哈希值和序列号来实现。 第二部分是解锁脚本，钱包构建它用以满足设定在UTXO中的支出条件。 大多数情况下，解锁脚本是一个证明比特币所有权的数字签名和公钥，但是并不是所有的解锁脚本都包含签名。 第三部分是序列号，稍后再讨论。

考虑我们在之前交易幕后章节提到的例子。交易输入是一个名为 vin 的数组（列表）：

```
"vin": [
  {
    "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
    "vout": 0,
    "scriptSig" : "3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c
4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL] 0484e
cc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457
eee41c04f4938de5cc17b4a10fa336a8d752adf",
    "sequence": 4294967295
  }
]
```

如您所见，列表中只有一个输入（因为一个UTXO包含足够的值来完成此付款）。 输入包含四个元素：

- 一个交易ID，引用包含正在使用的UTXO的交易
- 一个输出索引（vout），用于标识来自该交易的哪个UTXO被引用（第一个为零）
- 一个 scriptSig（解锁脚本），满足放置在UTXO上的条件，解锁它用于支出
- 一个序列号（稍后讨论）

在 Alice 的交易中，输入指向的交易ID是：

```
7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18
```

输出索引是0（即由该交易创建的第一个UTXO）。解锁脚本由Alice的钱包构建，首先检索引用的UTXO，检查其锁定脚本，然后使用它来构建所需的解锁脚本以满足此要求。

仅仅看这个输入，你可能已经注意到，除了对包含它引用的交易之外，我们无从了解这个UTXO的任何内容。我们不知道它的价值（多少satoshi金额），我们不知道设置支出条件的锁定脚本。要找到这些信息，我们必须通过检索整个交易来检索被引用的UTXO。请注意，由于输入的值未明确说明，因此我们还必须使用被引用的UTXO来计算在此交易中支付的费用（参见后面交易费用章节）。

不仅仅是Alice的钱包需要检索输入中引用的UTXO。一旦将该交易广播到网络，每个验证节点也将需要检索交易输入中引用的UTXO，以验证该交易。

因为缺乏语境，交易本身似乎不完整。他们在输入中引用UTXO，但是没有检索到UTXO，我们无法知道输入的值或其锁定条件。当编写比特币软件时，无论何时解码交易以验证它或计算费用或检查解锁脚本，您的代码首先必须从块链中检索引用的UTXO，以构建隐含但不存在于输入的UTXO引用中的语境。例如，要计算支付总额的交易费，您必须知道输入和输出值的总和。但是，如果不检索输入中引用的UTXO，则不知道它们的值。因此，在单个交易中计算交易费用的简单操作，实际上涉及多个交易的多个步骤和数据。

我们可以使用与比特币核心相同的命令序列，就像我们在检索Alice的交易（getrawtransaction和decodeawtransaction）时一样。因此，我们可以得到在前面的输入中引用的UTXO，并查看：

输入中引用的来自Alice 以前的交易中的UTXO：

```
"vout": [
   {
     "value": 0.10000000,
     "scriptPubKey": "OP_DUP OP_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP_EQ
UALVERIFY OP_CHECKSIG"
   }
 ]
```

我们看到这个UTXO的值为0.1BTC，并且它有一个包含“OP_DUP OP_HASH160 ...”的锁定脚本（scriptPubKey）。

> 小贴士：为了充分了解Alice的交易，我们必须检索引用以前的交易作为输入。 检索以前的交易和未花费的交易输出的函数是非常普遍的，并且存在于几乎每个比特币函数库和API中。

6.3.2.1交易序列化--交易输入

当交易被序列化以在网络上传输时，它们的输入被编码成字节流，如下表所示

Size	Field	Description
32 bytes	Transaction Hash	Pointer to the transaction containing the UTXO to be spent
4 bytes	Output Index	The index number of the UTXO to be spent; first one is 0
1–9 bytes (VarInt)	Unlocking-Script Size	Unlocking-Script length in bytes, to follow
Variable	Unlocking-Script	A script that fulfills the conditions of the UTXO locking script
4 bytes	Sequence Number	Used for locktime or disabled (0xFFFFFFFF)

与输出一样，我们来看看我们是否可以从序列化格式的 Alice 的交易中找到输入。 首先，将输入解码：

```
"vin":
[
  {
    "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
    "vout": 0,
    "scriptSig" : "3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c
4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL] 0484e
cc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457
eee41c04f4938de5cc17b4a10fa336a8d752adf",
    "sequence": 4294967295
  }
]
```

现在，我们来看看我们是否可以识别下面这些以十六进制表示法表示的字段：

0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd73
4d2804fe65fa35779000000008b483045022100884d142d86652a3f47
ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039
ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813
01410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade84
16ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc1
7b4a10fa336a8d752adfffffffff0260e31600000000001976a914ab6
8025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef800000000000
1976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac00000 000

提示：

- 交易ID以反转字节顺序序列化，因此以（十六进制）18开头，以79结尾
- 输出索引为4字节组的“0”，容易识别
- scriptSig的长度为139个字节，或十六进制为8b
- 序列号设置为FFFFFFFF，也容易识别

6.3.3 交易费

大多数交易包含交易费（矿工费），这是为了确保网络安全而给比特币矿工的一种补偿。费用本身也作为一个安全机制，使经济上不利于攻击者通过交易来淹没网络。对于挖矿、费用和矿工得到的奖励，在挖矿一章中将有更详细的讨论。

这一节解释交易费是如何被包含在一个典型的交易中的。大多数钱包自动计算并计入交易费。但是， 如果你以编程方式构造交易，或者使用命令行界面，你必须手动计算并计入这些费用。

交易费作为矿工打包（挖矿）一笔交易到下一个区块中的一种激励；同时作为一种抑制因素，通过对每一笔交易收取小额费用来防止对系统的滥用。成功挖到某区块的矿工将得到该区内包含的矿工费， 并将该区块添加至区块链中。

交易费是基于交易的千字节规模来计算的，而不是比特币交易的价值。总的来说，交易费是根据比特币网络中的市场力量确定的。矿工会依据许多不同的标准对交易进行优先级排序，包括费用，他们甚至可能在某些特定情况下免费处理交易。但大多数情况下，交易费影响处理优先级，这意味着有足够费用的交易会更可能被打包进下一个挖出的区块中；反之交易费不足或者没有交易费的交易可能会被推迟，基于尽力而为的原则在几个区块之后被处理，甚至可能根本不被处理。交易费不是强制的，而且没有交易费的交易最终也可能会被处理，但是，交易费将提高处理优先级。

随着时间的推移，交易费的计算方式以及在交易处理优先级上的影响已经产生了变化。起初，交易费是固定的，是网络中的一个固定常数。渐渐地，随着网络容量和交易量的不断变化，并可能受到来自市场力量的影响，收费结构开始放松。自从至少2016年初以来，比特币网络容量的限制已经造成交易之间的竞争，从而导致更高的费用，免费交易彻底成为过去式。零费用或非常低费用的交易鲜少被处理，有时甚至不会在网络上传播。

在 Bitcoin Core 中，费用传递政策由minrelaytxfee选项设置。 目前默认的minrelaytxfee是每千字节0.00001比特币或者millibitcoin的1%。 因此，默认情况下，费用低于0.0001比特币的交易是免费的，但只有在内存池有空间时才会被转发; 否则，会被丢弃。 比特币节点可以通过调整minrelaytxfee的值来覆盖默认的费用传策略。

任何创建交易的比特币服务，包括钱包，交易所，零售应用等，都必须实现动态收费。动态费用可以通过第三方费用估算服务或内置的费用估算算法来实现。如果您不确定，那就从第三方服务开始，如果您希望去除第三方依赖，您应当有设计和部署自己算法的经验。

费用估算算法根据网络能力和“竞争”交易提供的费用计算适当的费用。这些算法的范围从十分简单的（最后一个块中的平均值或中位数）到非常复杂的（统计分析）均有覆盖。他们估计必要的费用（以字节为单位），这将使得交易具有很高的可能性被选择并打包进一定数量的块内。大多数服务为用户提供高、中、低优先费用的选择。高优先级意味着用户支付更高的交易费，但交易可能就会被打包进下一个块中。中低优先级意味着用户支付较低的交易费，但交易可能需要更长时间才能确认。

许多钱包应用程序使用第三方服务进行费用计算。一个流行的服务是 http://bitcoinfees.21.co，它提供了一个API和一个可视化图表，以satoshi / byte为单位显示了不同优先级的费用。

> 小贴士：静态费用在比特币网络上不再可行。 设置静态费用的钱包将导致用户体验不佳，因为交易往往会被“卡住”，并不被确认。 不了解比特币交易和费用的用户因交易被“卡住” 而感到沮丧，因为他们认为自己已经失去了资金。

下面费用估算服务bitcoinfees.21.co中的图表显示了10个satoshi / byte增量的费用的实时估计，以及每个范围的费用交易的预期确认时间（分钟和块数）。 对于每个收费范围（例如，61-70 satoshi /字节），两个水平栏显示过去24小时（102,975）中未确认交易的数量（1405）和交易总数，费用在该范围内。 根据图表，此时推荐的高优先费用为80 satoshi / 字节，这可能导致交易在下一个块（零块延迟）中开采。 据合理判断，一笔常规交易的大小约为226字节，因此单笔交易建议费用为18,080 satoshis（0.00018080 BTC）。

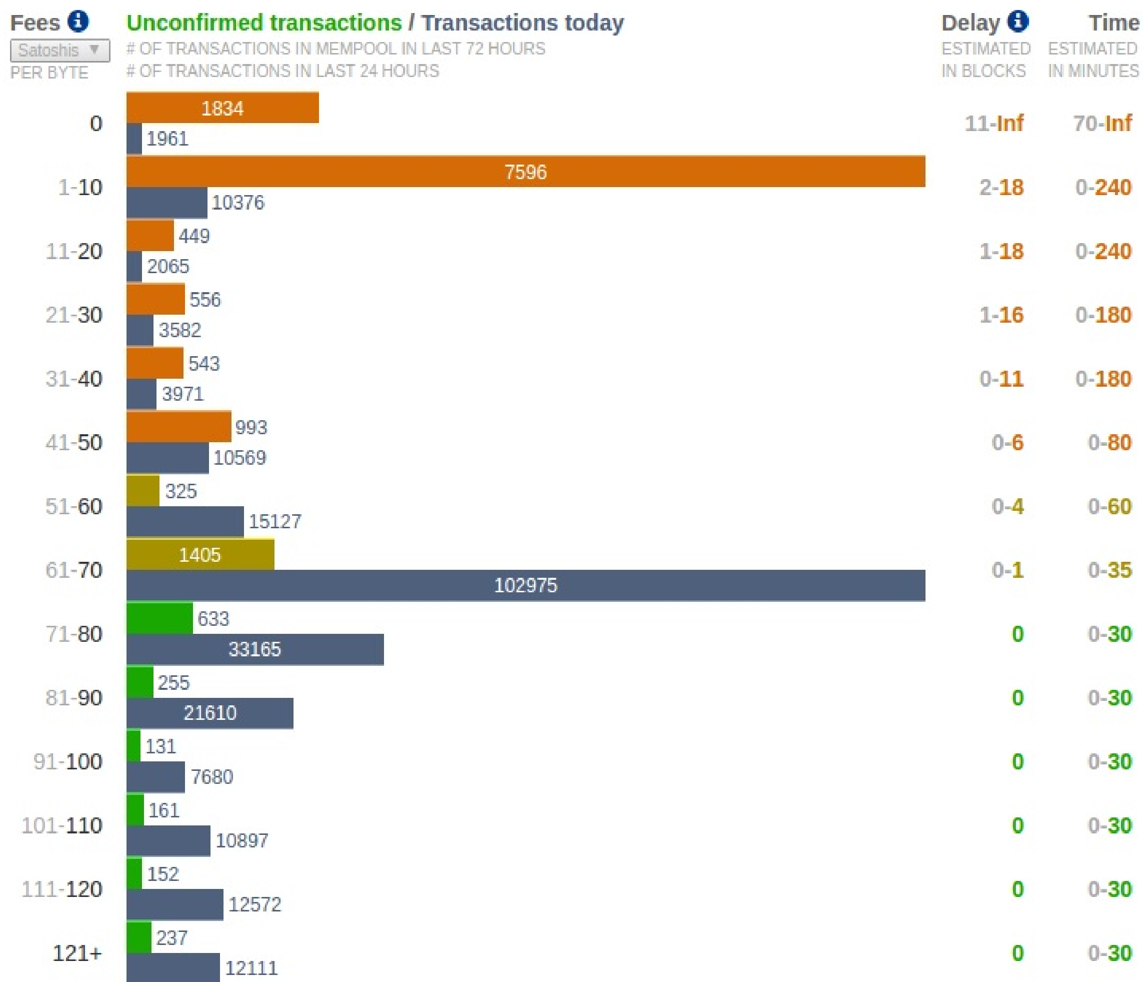
费用估算数据可以通过简单的HTTP REST API（https://bitcoinfees.21.co/api/v1/fees/recommended）来检索。 例如，在命令行中使用curl命令：

运用费用估算API

```
$ curl https://bitcoinfees.21.co/api/v1/fees/recommended

{"fastestFee":80,"halfHourFee":80,"hourFee":60}
```

API通过费用估算以 satoshi per byte 的方式返回一个 JSON 对象，从而实现”最快确认“ (fastestFee)，以及在三个块（halfHourFee）和六个块（hourFee）内确认。



6.3.4 把交易费加到交易中

交易的数据结构没有交易费的字段。相替代地，交易费是指输入和输出之间的差值。从所有输入中扣掉所有输出之后的多余的量会被矿工作为矿工费收集走：

> 交易费即输入总和减输出总和的余量：交易费 = 求和（所有输入） - 求和（所有输出）

正确理解交易比较困难，但又尤为重要。因为如果你要构建你自己的交易，你必须确保你没有因疏忽在交易中添加一笔大量交易费而大大减少了输入的可花费额。这意味着你必须计算所有的输入，如有必要则加上找零， 不然的话，结果就是你给了矿工一笔相当可观的劳务费！

举例来说，如果你消耗了一个20比特币的UTXO来完成1比特币的付款，你必须包含一笔19比特币的找零回到你的钱包。否则，那剩下的19比特币会被当作交易费，并将由挖出你交易的矿工收走。尽管你会得到高优先级的处理，并且让一个矿工喜出望外，但这很可能不是你想要的。

> 警告：如果你忘记了在手动构造的交易中增加找零的输出，系统会把找零当作交易费来处理。“不用找了！”也许不是你的真实意愿。

让我们重温一下Alice在咖啡店的交易来看看在实际中它如何运作。Alice想花0.015比特币购买咖啡。为了确保这笔交易能被立即处理，Alice想添加一笔交易费，比如说0.001。这意味着总花费会变成0.016。因此她的钱包需要凑齐 0.016或更多的UTXO。如果需要，还要加上找零。我们假设他的钱包有一个0.2比特币的UTXO可用。他的钱包就会消耗 掉这个UTXO，创造一个新的0.015的输出给Bob的咖啡店，另一个0.184比特币的输出作为找零回到Alice的钱包， 并留下未分配的0.001矿工费内含在交易中。

现在让我们看看另一种情况。Eugenia，我们在菲律宾的儿童募捐项目主管，完成了一次为孩子购买教材的筹款活动。她从世界范围内接收到了好几千份小额捐款，总额是50比特币。所以她的钱包塞满了非常小的UTXO。现在她想用比特币从本地的一家出版商购买几百本教材。

现在Eugenia的钱包应用想要构造一个单笔大额付款交易，它必须从可用的、由很多小数额构成的大的UTXO集合中寻求钱币来源。这意味着交易的结果是从上百个小额UTXO中作为输入，但只有一个输出用来付给出版商。输入数量这么巨大的交易会比一千字节要大，也许总尺寸会达到两至三千字节。结果是它将需要比中等规模交易要高得多的交易费。

Eugenia的钱包应用会通过测量交易的大小，乘以每千字节需要的费用来计算适当的交易费。很多钱包会支付较大的交易费，确保交易得到及时处理。更高交易费不是因为Eugenia付的钱很多，而是因为她的交易很复杂并且尺寸更大——交易费是与参加交易的比特币值无关的。

6.4比特币交易脚本和脚本语言

比特币交易脚本语言，称为脚本，是一种类似Forth的逆波兰表达式的基于堆栈的执行语言。如果听起来不知所云，是你可能还没有学习20世纪60年代的编程语言，但是没关系，我们将在本章中解释这一切。 放置在UTXO上的锁定脚本和解锁脚本都以此脚本语言编写。 当一笔比特币交易被验证时，每一个输入值中的解锁脚本与其对应的锁定脚本同时 （互不干扰地）执行，以确定这笔交易是否满足支付条件。

脚本是一种非常简单的语言，被设计为在执行范围上有限制，可在一些硬件上执行，可能与嵌入式装置一样简单。 它仅需要做最少的处理，许多现代编程语言可以做的花哨的事情它都不能做。 但用于验证可编程货币，这是一个经深思熟虑的安全特性。

如今，大多数经比特币网络处理的交易是以“Alice付给Bob”的形式存在，并基于一种称为“P2PKH”（Pay-toPublic-Key-Hash）脚本。但是，比特币交易不局限于“支付给Bob的比特币地址”的脚本。事实上，锁定脚本可以被编写成表达各种复杂的情况。为了理解这些更为复杂的脚本，我们必须首先了解交易脚本和脚本语言的基础知识。

在本节中，我们将会展示比特币交易脚本语言的各个组件；同时，我们也会演示如何使用它去表达简单的使用条件以及如何通过解锁脚本去满足这些花费条件。

> 小贴士：比特币交易验证并不基于静态模式，而是通过脚本语言的执行来实现的。这种语言允许表达几乎无限的各种条件。这也是比特币作为一种“可编程的货币”所拥有的力量。

6.4.1 图灵非完备性

比特币脚本语言包含许多操作码，但都故意限定为一种重要的模式——除了有条件的流控制以外，没有循环或复杂流控制能力。这样就保证了脚本语言的图灵非完备性，这意味着脚本有限的复杂性和可预见的执行次数。脚本并不是一种通用语言，这些限制确保该语言不被用于创造无限循环或其它类型的逻辑炸弹，这样的炸弹可以植入在一笔交易中，引起针对比特币网络的“拒绝服务”攻击。记住，每一笔交易都会被网络中的全节点验证，受限制的语言能防止交易验证机制被作为一个漏洞而加以利用。

6.4.2 去中心化验证

比特币交易脚本语言是没有中心化主体的，没有任何中心主体能凌驾于脚本之上，也没有中心主体会在脚本被执行后对其进行保存。所以执行脚本所需信息都已包含在脚本中。可以预见的是，一个脚本能在任何系统上以相同的方式执行。如果您的系统验证了一个脚本，可以确信的是每一个比特币网络中的其他系统也将验证这个脚本，这意味着一个有效的交易对每个人而言都是有效的，而且每一个人都知道这一点。这种结果可预见性是比特币系统的一项至关重要的良性特征。

6.4.3 脚本构建（锁定与解锁）

比特币的交易验证引擎依赖于两类脚本来验证比特币交易：锁定脚本和解锁脚本。

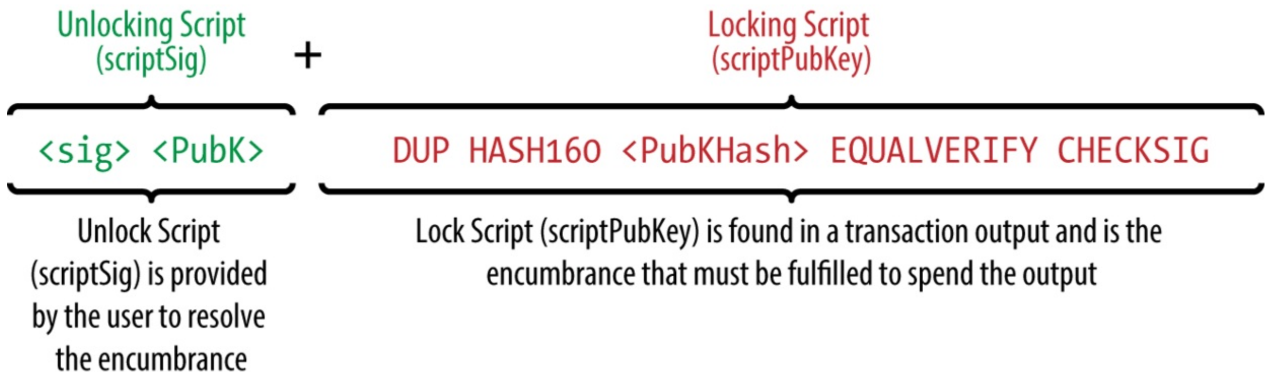
锁定脚本是一个放置在输出上面的花费条件：它指定了今后花费这笔输出必须要满足的条件。 由于锁定脚本往往含有一个公钥或比特币地址（公钥哈希值），在历史上它曾被称为脚本公钥（scriptPubKey）。由于认识到这种脚本技术存在着更为广泛的可能性，在本书中，我们将它称为“锁定脚本”（locking script）。在大多数比特币应用程序中，我们所称的“锁定脚本”将以scriptPubKey的形式出现在源代码中。您还将看到被称为见证脚本（witness script）的锁定脚本（参见[隔离见证]章节），或者更一般地说，它是一个加密难题（cryptographic puzzle）。 这些术语在不同的抽象层次上都意味着同样的东西。

解锁脚本是一个“解决”或满足被锁定脚本在一个输出上设定的花费条件的脚本，它将允许输出被消费。解锁脚本是每一笔比特币交易输入的一部分，而且往往含有一个由用户的比特币钱包（通过用户的私钥）生成的数字签名。由于解锁脚本常常包含一个数字签名，因此它曾被称作ScriptSig。在大多数比特币应用的源代码中，ScriptSig便是我们所说的解锁脚本。你也会看到解锁脚本被称作“见证”（witness 参见[隔离见证]章节）。在本书中，我们将它称为“解锁脚本”，用以承认锁定脚本的需求有更广的范围。但并非所有解锁脚本都一定会包含签名。

每一个比特币验证节点会通过同时执行锁定和解锁脚本来验证一笔交易。每个输入都包含一个解锁脚本，并引用了之前存在的UTXO。 验证软件将复制解锁脚本，检索输入所引用的UTXO，并从该UTXO复制锁定脚本。 然后依次执行解锁和锁定脚本。 如果解锁脚本满足锁定脚本条件，则输入有效（请参阅单独执行解锁和锁定脚本部分）。 所有输入都是独立验证的，作为交易总体验证的一部分。

请注意，UTXO被永久地记录在区块链中，因此是不变的，并且不受在新交易中引用失败的尝试的影响。 只有正确满足输出条件的有效交易才能将输出视为“开销来源”，继而该输出将被从未花费的交易输出集（UTXO set）中删除。

下图是最常见类型的比特币交易（P2PKH:对公钥哈希的付款）的解锁和锁定脚本的示例，显示了在脚本验证之前从解锁和锁定脚本的并置产生的组合脚本：



6.4.3.1脚本执行堆栈

比特币的脚本语言被称为基于堆栈的语言，因为它使用一种被称为堆栈的数据结构。堆栈是一个非常简单的数据结构，可以被视为一叠卡片。栈允许两个操作：push和pop（推送和弹出）。 Push（推送）在堆栈顶部添加一个项目。 Pop（弹出）从堆栈中删除最顶端的项。栈上的操作只能作用于栈最顶端项目。堆栈数据结构也被称为“后进先出”（ Last-In-First-Out）或 “LIFO” 队列。

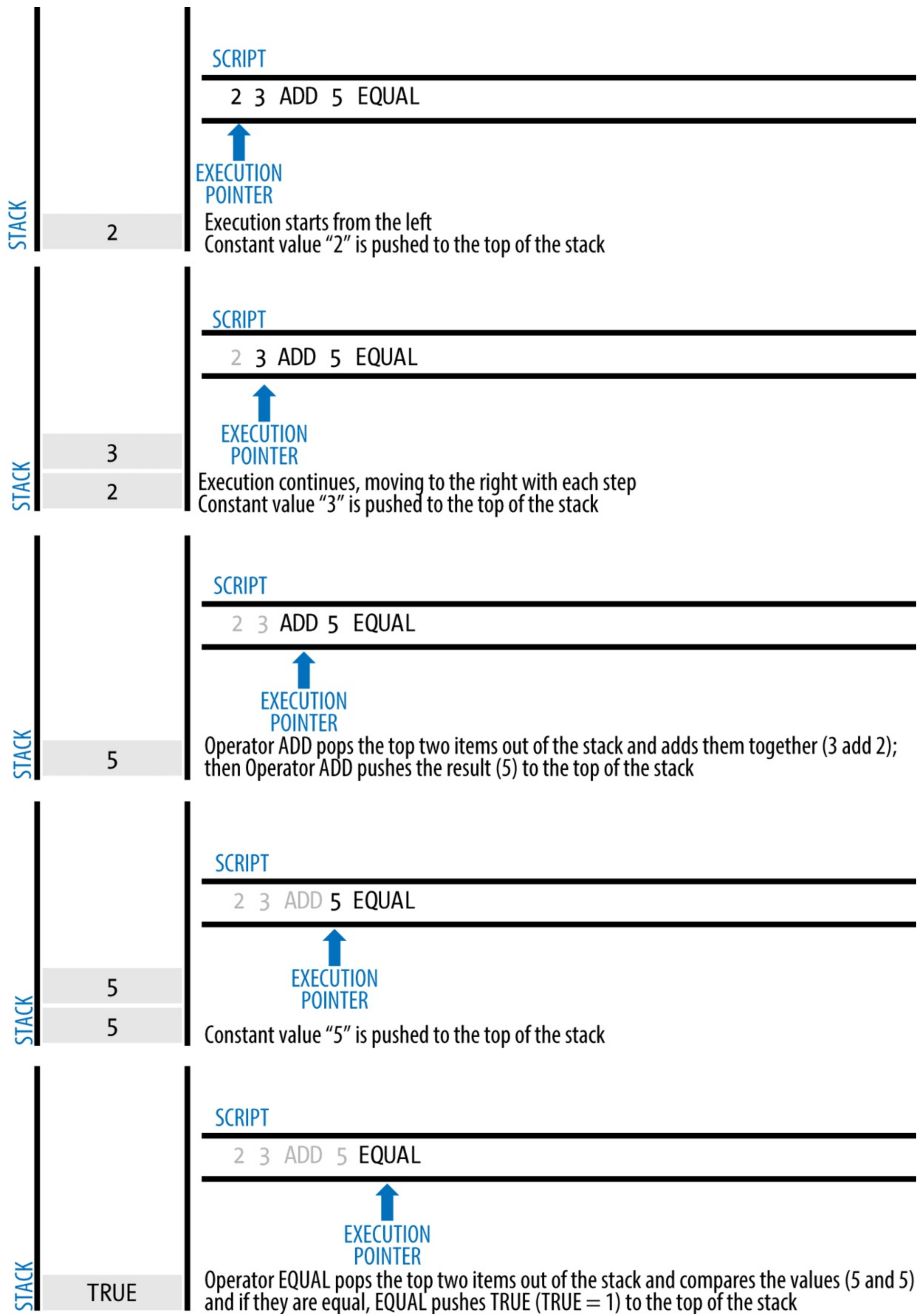
脚本语言通过从左到右处理每个项目来执行脚本。数字（数据常量）被推到堆栈上。操作码（Operators）从堆栈中推送或弹出一个或多个参数，对其进行操作，并可能将结果推送到堆栈上。例如，操作码 OP_ADD 将从堆栈中弹出两个项目，添加它们，并将结果的总和推送到堆栈上。

条件操作码（Conditional operators）对一个条件进行评估，产生一个 TRUE 或 FALSE 的布尔结果（boolean result）。例如， OP_EQUAL 从堆栈中弹出两个项目，如果它们相等，则推送为 TRUE（由数字1表示），否则推送为 FALSE（由数字0表示）。比特币交易脚本通常包含条件操作码，以便它们可以产生用来表示有效交易的 TRUE 结果。

6.4.3.2一个简单的脚本

现在让我们将学到的关于脚本和堆栈的知识应用到一些简单的例子中。

如图6-4，在比特币的脚本验证中，执行简单的数学运算时，脚本“ 2 3 OP*ADD 5 OP_EQUAL* ”演示了算术加法操作码 *OP_ADD* ，该操作码将两个数字相加，然后把结果推送到堆栈， 后面的条件操作符 *OP_EQUAL* 是验算之前的两数之和是否等于 *5* 。为了简化起见，前缀*OP*在一步步的演示过程中将被省略。有关可用脚本操作码和函数的更多详细信息，请参见[交易脚本]。



SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
2
Execution starts from the left
Constant value "2" is pushed to the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
3
2
Execution continues, moving to the right with each step
Constant value "3" is pushed to the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
5
Operator ADD pops the top two items out of the stack and adds them together (3 add 2);
then Operator ADD pushes the result (5) to the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
5
5
Constant value "5" is pushed to the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
TRUE
Operator EQUAL pops the top two items out of the stack and compares the values (5 and 5)
and if they are equal, EQUAL pushes TRUE (TRUE = 1) to the top of the stack

尽管绝大多数解锁脚本都指向一个公钥哈希值（本质上就是比特币地址），因此如果想要使用资金则需验证所有权，但脚本本身并不需要如此复杂。任何解锁和锁定脚本的组合如果结果为真（TRUE），则为有效。前面被我们用于说明脚本语言的简单算术操作码同样也是一个有效的锁定脚本，该脚本能用于锁定交易输出。

使用部分算术操作码脚本作为锁定脚本的示例：

```
3 OP_ADD 5 OP_EQUAL
```

该脚本能被以如下解锁脚本为输入的一笔交易所满足：

```
2
```

验证软件将锁定和解锁脚本组合起来，结果脚本是：

```
2 3 OP_ADD 5 OP_EQUAL
```

正如在上图中所看到的，当脚本被执行时，结果是OP_TRUE，交易有效。不仅该笔交易的输出锁定脚本有效，同时UTXO也能被任何知晓这个运算技巧（知道是数字2）的人所使用。

> 小贴士：如果堆栈顶部的结果显示为TRUE（标记为{{0x01}}），即为任何非零值，或脚本执行后堆栈为空情形，则交易有效。如果堆栈顶部的结果显示为FALSE（0字节空值，标记为{{}}）或脚本执行被操作码明确禁止，如OP_VERIFY、 OP_RETURN，或有条件终止如OP_ENDIF，则交易无效。详见[tx_script_ops]相关内容。

以下是一个稍微复杂一点的脚本，它用于计算 2+7-3+1 。注意，当脚本在同一行包含多个操作码时，堆栈允许一个操作码的结果由于下一个操作码执行。

```
2 7 OP_ADD 3 OP_SUB 1 OP_ADD 7 OP_EQUAL
```

请试着用纸笔自行演算脚本，当脚本执行完毕时，你会在堆栈得到正确的结果。

6.4.3.3 解锁和锁定脚本的单独执行

在最初版本的比特币客户端中，解锁和锁定脚本是以连锁的形式存在，并被依次执行的。出于安全因素考虑，在2010 年比特币开发者们修改了这个特性——因为存在“允许异常解锁脚本推送数据入栈并且污染锁定脚本”的漏洞。而在当前的方案中，这两个脚本是随着堆栈的传递被分别执行的。下面将会详细介绍。

首先，使用堆栈执行引擎执行解锁脚本。如果解锁脚本在执行过程中未报错（例如：没有“悬挂”操作码），则复制主堆栈（而不是备用堆栈），并执行锁定脚本。如果从解锁脚本中复制而来的堆栈数据执行锁定脚本的结果为“TRUE"，那么解锁脚本就成功地满足了锁定脚本所设

置的条件，因此，该输入是一个能使用该UTXO的有效授权。如果在合并脚本后的结果不是"TRUE"以外的任何结果，输入都是无效的，因为它不能满足UTXO中所设置的使用该笔资金的条件。

6.4.4 P2PKH（Pay-to-Public-Key-Hash）

比特币网络处理的大多数交易花费的都是由"付款至公钥哈希"（或P2PKH）脚本锁定的输出，这些输出都含有一个锁定脚本，将输入锁定为一个公钥哈希值，即我们常说的比特币地址。由P2PKH脚本锁定的输出可以通过提供一个公钥和由相应私钥创建的数字签名来解锁（使用）。参见数字签名ECDSA相关内容。

例如，我们可以再次回顾一下Alice向Bob咖啡馆支付的案例。Alice下达了向Bob咖啡馆的比特币地址支付0.015比特币的支付指令，该笔交易的输出内容为以下形式的锁定脚本：

```
OP_DUP OP_HASH160 <Cafe Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG
```

脚本中的 Cafe Public Key Hash 即为咖啡馆的比特币地址，但该地址不是基于Base58Check编码。事实上，大多数比特币地址的公钥哈希值都显示为十六进制码，而不是大家所熟知的以1开头的基于Bsase58Check编码的比特币地址。

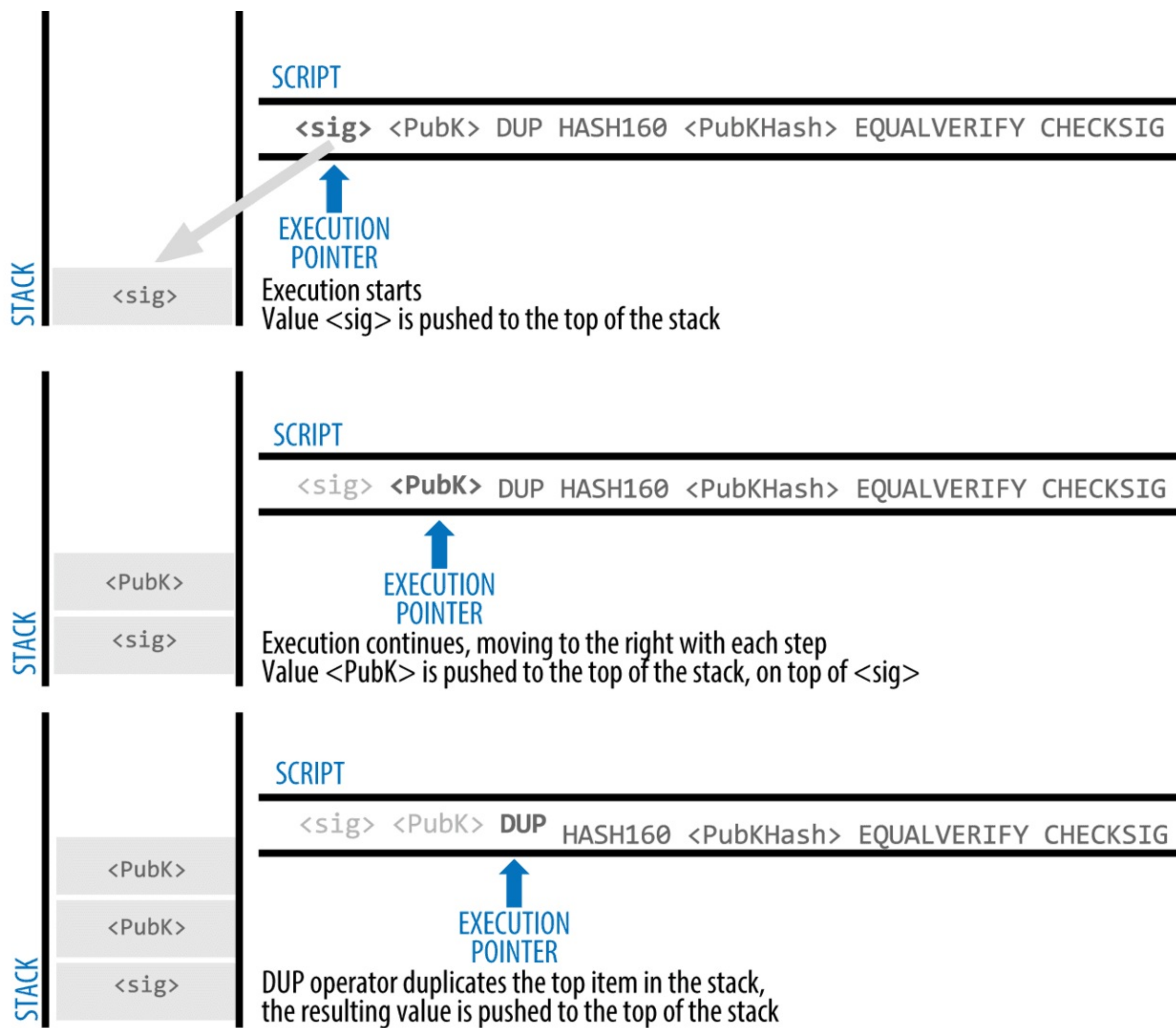
上述锁定脚本相应的解锁脚本是：

```
<Cafe Signature> <Cafe Public Key>
```

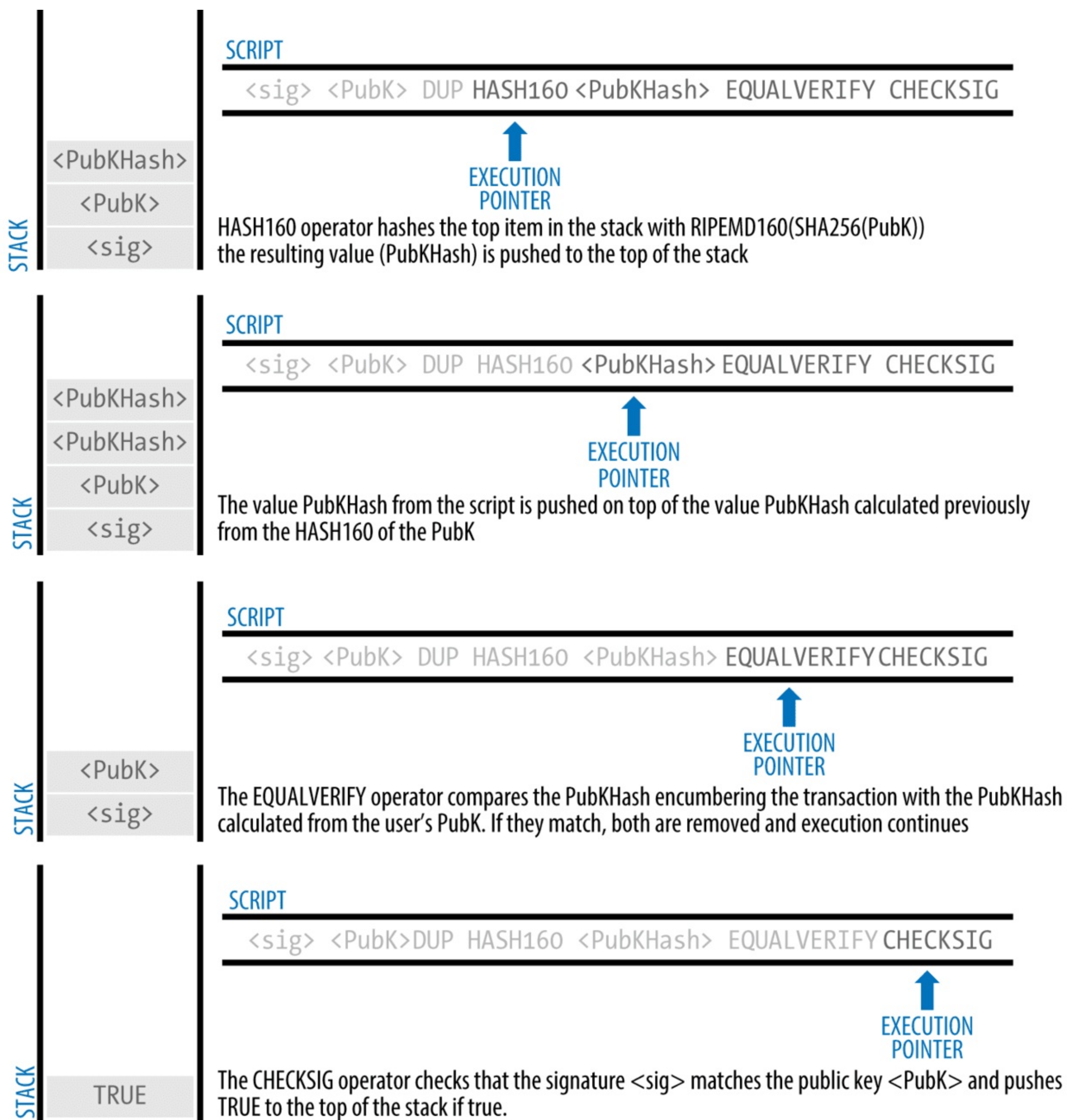
将两个脚本结合起来可以形成如下组合验证脚本：

```
<Cafe Signature> <Cafe Public Key> OP_DUP OP_HASH160
<Cafe Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG
```

只有当解锁脚本与锁定脚本的设定条件相匹配时，执行组合验证脚本时才会显示结果为真（TRUE）。换句话说，只有当解锁脚本得到了咖啡馆的有效签名，交易执行结果才会被通过（结果为真），该有效签名是从与公钥哈希相匹配的咖啡馆的私钥中所获取的。 图6-5和图6-6（分两部分）显示了组合脚本一步步检验交易有效性的过程。



SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
STACK
<sig>
Execution starts
Value <sig> is pushed to the top of the stack
SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
STACK
<PubK>
<sig>
Execution continues, moving to the right with each step
Value <PubK> is pushed to the top of the stack, on top of <sig>
SCRIPT
<sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
EXECUTION POINTER
STACK
<PubK>
<PubK>
<sig>
DUP operator duplicates the top item in the stack,
the resulting value is pushed to the top of the stack



6.5数字签名（ECDSA）

到目前为止，我们还没有深入了解“数字签名”的细节。在本节中，我们将研究数字签名的工作原理，以及如何在不揭示私钥的情况下提供私钥的所有权证明。

比特币中使用的数字签名算法是椭圆曲线数字签名算法（Elliptic Curve Digital Signature Algorithm）或ECDSA。 ECDSA是用于基于椭圆曲线私钥/公钥对的数字签名的算法，如椭圆曲线章节[elliptic_curve]所述。 ECDSA用于脚本函数OP_CHECKSIG，OP_CHECKSIGVERIFY，OP_CHECKMULTISIG和OP_CHECKMULTISIGVERIFY。每当你锁定脚本中看到这些时，解锁脚本都必须包含一个ECDSA签名。

数字签名在比特币中有三种用途（请参阅下面的侧栏）。第一，签名证明私钥的所有者，即资金所有者，已经授权支出这些资金。第二，授权证明是不可否认的（不可否认性）。第三，签名证明交易（或交易的具体部分）在签字之后没有也不能被任何人修改。

请注意，每个交易输入都是独立签名的。这一点至关重要，因为不管是签名还是输入都不必由同一“所有者”实施。事实上，一个名为 “CoinJoin” 的特定交易方案（多重签名方案？）就使用这个特性来创建多方交易来保护隐私。

> 注意：每个交易输入和它可能包含的任何签名完全独立于任何其他输入或签名。 多方可以协作构建交易，并各自仅签一个输入。
>
> 维基百科对 “数字签名 ”的定义：
>
> 数字签名是用于证明数字消息或文档的真实性的数学方案。 有效的数字签名给了一个容易接受的理由去相信：*1*）该消息是由已知的发送者（身份认证性）创建的； *2*）发送方不能否认已发送消息（不可否认性；*3*）消息在传输中未被更改（完整性）。

来源: https://en.wikipedia.org/wiki/Digital_signature*

6.5.1数字签名如何工作

数字签名是一种由两部分组成的数学方案：第一部分是使用私钥（签名密钥）从消息（交易）创建签名的算法； 第二部分是允许任何人验证签名的算法，给定消息和公钥。

6.5.1.1创建数字签名

在比特币的ECDSA算法的实现中，被签名的“消息”是交易，或更确切地说是交易中特定数据子集的哈希值（参见签名哈希类型（SIGHASH））。签名密钥是用户的私钥，结果是签名：

((Sig = F{sig}(F{hash}(m), dA)))

这里的：

- *dA* 是签名私钥
- *m* 是交易（或其部分）
- *F~hash~* 是散列函数
- *F~sig~* 是签名算法
- *Sig* 是结果签名

ECDSA数学运算的更多细节可以在ECDSA Math章节中找到。

函数*F~sig~* 产生由两个值组成的签名*Sig*，通常称为*R*和*S*：

```
Sig = (R, S)
```

现在已经计算了两个值R和S，它们就序列化为字节流，使用一种称为“分辨编码规则”（*Distinguished Encoding Rules*）或 DER的国际标准编码方案。

6.5.1.2签名序列化（DER）

我们再来看看Alice创建的交易。 在交易输入中有一个解锁脚本，其中包含Alice的钱包中的以下DER编码签名：

```
3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039f
f08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e381301
```

该签名是Alice的钱包生成的R和S值的序列化字节流，证明她拥有授权花费该输出的私钥。 序列化格式包含以下9个元素：

- *0x30*表示DER序列的开始
- *0x45* - 序列的长度（69字节）
- *0x02* - 一个整数值
- *0x21* - 整数的长度（33字节）
- *R-00884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb*
- *0x02* - 接下来是一个整数
- *0x20* - 整数的长度（32字节）
- *S-4b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813*
- 后缀（*0x01*）指示使用的哈希的类型（SIGHASH_ALL）

看看您是否可以使用此列表解码 Alice 的序列化（DER编码）签名。 重要的数字是R和S; 数据的其余部分是DER编码方案的一部分。

6.5.2验证签名

要验证签名，必须有签名（*R*和*S*）、序列化交易和公钥（对应于用于创建签名的私钥）。本质上，签名的验证意味着“只有生成此公钥的私钥的所有者，才能在此交易上产生此签名。”

签名验证算法采用消息（交易或其部分的哈希值）、签名者的公钥和签名（R和S值），如果签名对该消息和公钥有效，则返回 TRUE 值。

6.5.3签名哈希类型（SIGHASH）

数字签名被应用于消息，在比特币中，就是交易本身。签名意味着签字人对特定交易数据的承诺（ *commitment*）。在最简单的形式中，签名适用于整个交易，从而承诺（commit）所有输入，输出和其他交易字段。但是，在一个交易中一个签名可以只承诺（*commit*）一个数据子集，这对于我们将在本节中看到的许多场景是有用的。

比特币签名具有指示交易数据的哪一部分包含在使用 SIGHASH 标志的私钥签名的哈希中的方式。 SIGHASH 标志是附加到签名的单个字节。每个签名都有一个SIGHASH标志，该标志在不同输入之间也可以不同。具有三个签名输入的交易可以具有不同SIGHASH标志的三个签名，每个签名签署（承诺）交易的不同部分。

记住，每个输入可能在其解锁脚本中包含一个签名。因此，包含多个输入的交易可以拥有具有不同SIGHASH标志的签名，这些标志在每个输入中承诺交易的不同部分。还要注意，比特币交易可能包含来自不同“所有者”的输入，他们在部分构建（和无效）的交易中可能仅签署一个输入，继而与他人协作收集所有必要的签名后再使交易生效。许多SIGHSASH标志类型，只有在你考虑到由许多参与者在比特币网络之外共同协作去更新仅部分签署了的交易，才具有意义。

有三个SIGHASH标志：ALL，NONE和SINGLE，如下表所示。

SIGHASH flag	Value	Description
ALL	0x01	Signature applies to all inputs and outputs
NONE	0x02	Signature applies to all inputs, none of the outputs
SINGLE	0x03	Signature applies to all inputs but only the one output with the same index number as the signed input

另外还有一个修饰符标志SIGHASH_ANYONECANPAY，它可以与前面的每个标志组合。 当设置ANYONECANPAY时，只有一个输入被签名，其余的（及其序列号）打开以进行修改。ANYONECANPAY的值为0x80，并通过按位OR运算，得到如下所示的组合标志：

SIGHASH flag	Value	Description
ALL|ANYONECANPAY	0x81	Signature applies to one inputs and all outputs
NONE|ANYONECANPAY	0x82	Signature applies to one inputs, none of the outputs
SINGLE|ANYONECANPAY	0x83	Signature applies to one input and the output with the same index number

SIGHASH标志在签名和验证期间应用的方式是建立交易的副本和删节其中的某些字段（设置长度为零并清空），继而生成的交易被序列化，SIGHASH标志被添加到序列化交易的结尾，并将结果哈希化 ，得到的哈希值本身即是被签名的“消息”。 基于SIGHASH标志的使用，交易的不同部分被删节。 所得到的哈希值取决于交易中数据的不同子集。 在哈希化前，SIGHASH作为最后一步被包含在内，签名也会对SIGHASH类型进行签署，因此不能更改（例如，被矿工）。

> 小贴士：所有SIGHASH类型对应交易nLocktime字段（请参阅[transaction_locktime_nlocktime]部分）。 此外，SIGHASH类型本身在签名之前附加到交易，因此一旦签名就不能修改它。

在Alice的交易（参见序列化签名（DER）的列表）的例子中，我们看到DER编码签名的最后一部分是01，这是SIGHASH_ALL标志。这会锁定交易数据，因此Alice的签名承诺的是所有的输入和输出状态。 这是最常见的签名形式。

我们来看看其他一些SIGHASH类型，以及如何在实践中使用它们：

ALL | ANYONECANPAY

这种构造可以用来做“众筹”交易，试图筹集资金的人可以用单笔输出来构建一个交易，单笔输出将“目标”金额付给众筹发起人。这样的交易显然是无效的，因为它没有输入。但是现在其他人可以通过添加自己的输入作为捐赠来修改它们，他们用ALL | ANYONECANPAY签署自己的输入，除非收集到足够的输入以达到输出的价值，交易无效，每次捐赠是一项“抵押”，直到募集整个目标金额才能由募款人收取。

NONE

该结构可用于创建特定数量的“不记名支票”或“空白支票”。它对输入进行承诺，但允许输出锁定脚本被更改。任何人都可以将自己的比特币地址写入输出锁定脚本并兑换交易。然而，输出值本身被签名锁定。

NONE | ANYONECANPAY

这种构造可以用来建造一个“吸尘器”。在他们的钱包中拥有微小UTXO的用户无法花费这些费用，因为手续费用超过了这些微小UTXO的价值。借助这种类型的签名，微小UTXO可以为任何人捐赠，以便随时随地收集和消费。

有一些修改或扩展SIGHASH系统的建议。作为Elements项目的一部分，一个这样的提案是Blockstream的Glenn Willen提出的Bitmask Sighash模式。这旨在为SIGHASH类型创建一个灵活的替代品，允许“任意的，输入和输出的矿工可改写位掩码”来表示“更复杂的合同预付款方案，例如已分配的资产交换中有变更的已签名的报价”。

> 注释: 您不会在用户的钱包应用程序中看到SIGHASH标志作为一个功能呈现。 少数例外，钱包会构建P2PKH脚本，并使用SIGHASH_ALL标志进行签名。 要使用不同的SIGHASH标志，您必须编写软件来构造和签署交易。 更重要的是，SIGHASH标志可以被专用的比特币应用程序使用，从而实现新颖的用途。

6.5.4 ECDSA数学

如前所述，签名由产生由两个值*R*和*S*组成的签名的数学函数*F~sig~* 创建。在本节中，我们将查看函数*F~sig~* 的更多细节。

签名算法首先生成一个 *ephemeral*（临时）私公钥对。 在涉及签名私钥和交易哈希的变换之后，该临时密钥对用于计算*R*和*S*值。

临时密钥对基于随机数k，用作临时私钥。 从*k*，我们生成相应的临时公钥*P*（以P = k *G*计算，与派生比特币公钥相同）;参见*[pubkey]*部分）。数字签名的R值则是临时公钥P的x*坐标。

从那里，算法计算签名的S值，使得：

$S = k^{-1} (Hash(m) + dA * R) \bmod p$

其中：

- *k*是临时私钥
- *R*是临时公钥的x坐标
- *dA*是签名私钥
- *m*是交易数据
- *p*是椭圆曲线的主要顺序

验证是签名生成函数的倒数，使用R，S值和公钥来计算一个值P，该值是椭圆曲线上的一个点（签名创建中使用的临时公钥）：

$P = S^{-1} * Hash(m) * G + S^{-1} * R * Qa$

其中：

- *R*和*S*是签名值
- *Qa*是Alice的公钥
- *m*是签署的交易数据
- *G*是椭圆曲线发生器点

如果计算点*P*的x坐标等于*R*，则验证者可以得出结论，签名是有效的。

请注意，在验证签名时，私钥既不知道也不显示。

> 小贴士：ECDSA的数学很复杂，难以理解。 网上有一些很棒的指南可能有帮助。 搜索“ECDSA解释”或尝试这个：http://bit.ly/2r0HhGB。

6.5.5随机性在签名中的重要性

如我们在ECDSA Math中所看到的，签名生成算法使用随机密钥*k*作为临时私有-公钥对的基础。 *k* 的值不重要，只要它是随机的。如果使用相同的值 *k* 在不同的消息（交易）上产生两个签名，那么签名私钥可以由任何人计算。在签名算法中重用相同的 *k* 值会导致私钥的暴露！

> 警告 如果在两个不同的交易中，在签名算法中使用相同的值 *k*，则私钥可以被计算并暴露给世界！

这不仅仅是一个理论上的可能性。我们已经看到这个问题导致私人密钥在比特币中的几种不同实现的交易签名算法中的暴露。人们由于无意中重复使用 *k* 值而将资金窃取。重用 *k* 值的最常见原因是未正确初始化的随机数生成器。

为了避免这个漏洞，业界最佳实践不是用熵播种的随机数生成器生成 *k* 值，而是使用交易数据本身播种的确定性随机进程。这确保每个交易产生不同的 *k* 值。在互联网工程任务组（Internet Engineering Task Force）发布的RFC 6979中定义了 *k* 值的确定性初始化的行业标准算法。

如果您正在实现一种用于在比特币中签署交易的算法，则必须使用RFC 6979或类似的确定性随机算法来确保为每个交易生成不同的 *k* 值。

6.6比特币地址，余额和其他摘要

在本章开始，我们发现交易的“幕后”看起来与它在钱包、区块链浏览器和其它面向用户的应用程序中呈现的非常不同。 来自前几章的许多简单而熟悉的概念，如比特币地址和余额，似乎在交易结构中不存在。 我们看到交易本身并不包含比特币地址，而是通过锁定和解锁比特币离散值的脚本进行操作。 这个系统中的任何地方都不存在余额，而每个钱包应用程序都明明白白地显示了用户钱包的余额。

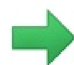
现在我们已经探讨了一个比特币交易中实际包含的内容，我们可以检查更高层次的抽象概念是如何从交易的看似原始的组成部分中派生出来的。

我们再来看看Alice的交易是如何在一个受欢迎的区块浏览器（前面章节Alice与Bob's Cafe的交易）中呈现的：

Transaction View information about a bitcoin transaction

0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2

1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK (0.1 BTC - Output)



1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA - (Unspent) 0.015 BTC
1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK - (Unspent) 0.0845 BTC

97 Confirmations | 0.0995 BTC

Summary	
Size	258 (bytes)
Received Time	2013-12-27 23:03:05
Included In Blocks	277316 (2013-12-27 23:11:54 +9 minutes)

Inputs and Outputs	
Total Input	0.1 BTC
Total Output	0.0995 BTC
Fees	0.0005 BTC
Estimated BTC Transacted	0.015 BTC

在交易的左侧，区块浏览器将Alice的比特币地址显示为“发送者”。其实这个信息本身并不在交易中。当区块链接浏览器检索到交易时，它还检索在输入中引用的先前交易，并从该旧交易中提取第一个输出。在该输出内是一个锁定脚本，将UTXO锁定到Alice的公钥哈希（P2PKH脚本）。块链浏览器提取公钥哈希，并使用Base58Check编码对其进行编码，以生成和显示表示该公钥的比特币地址。

同样，在右侧，区块浏览器显示了两个输出;第一个到Bob的比特币地址，第二个到Alice的比特币地址（作为找零）。再次，为了创建这些比特币地址，区块链浏览器从每个输出中提取锁定脚本，将其识别为P2PKH脚本，并从内部提取公钥哈希。最后，块链浏览器重新编码了使用Base58Check的公钥哈希生成和显示比特币地址。

如果您要点击Bob的比特币地址，则块链接浏览器将显示Bob的比特币地址的余额：

Bitcoin Address Addresses are identifiers which you use to send bitcoins to another person.

Summary	
Address	1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA
Hash 160	ab68025513c3dbd2f7b92a94e0581f5d50f654e7
Tools	Taint Analysis - Related Tags - Unspent Outputs

Transactions	
No. Transactions	25
Total Received	0.17579525 BTC
Final Balance	0.17579525 BTC

区块链浏览器显示了Bob的比特币地址的余额。但是比特币系统中却没有“余额”的概念。这么说吧，这里显示的余额其实是由区块链游览器按如下方式构建出来的：

为了构建“总接收”数量，区块链浏览器首先解码比特币地址的Base58Check编码，以检索编码在地址中的Bob的公钥的160位哈希值。然后，区块链浏览器搜索交易数据库，使用包含Bob公钥哈希的P2PKH锁定脚本寻找输出。通过总结所有输出的值，浏览器可以产生接收的总值。

完成构建当前余额（显示为“最终余额”）需要更多的工作。区块链接浏览器将当前未被使用的输入保存为一个分离的数据库——UTXO集。为了维护这个数据库，区块链浏览器必须监视比特币网络，添加新创建的UTXO，并在已被使用的UTXO出现在未经确认的交易中时，实时地删除它们。这是一个复杂的过程，不但要实时地跟踪交易在网络上的传播，同时还要保持与比特币网络的共识，确保在正确的链上。有时区块链浏览器未能保持同步，导致其对UTXO集的跟踪扫描不完整或不正确。

通过计算UTXO集，区块链浏览器总结了引用Bob的公钥哈希的所有未使用输出的值，并产生向用户显示的“最终余额”数目。

为了生成这张图片，得到这两个“余额”，区块链浏览器必须索引并搜索数十、数百甚至数十万的交易。

总之，通过钱包应用程序、区块链浏览器和其他比特币用户界面呈现给用户的信息通常源于更高层次的，通过搜索许多不同的交易，检查其内容以及操纵其中包含的数据而的抽象而构成。为了呈现出比特币交易类似于银行支票从发送人到接收人的这种简单视图，这些应用程

序必须抽象许多底层细节。他们主要关注常见的交易类型：每个输入上具有SIGHASH_ALL签名的P2PKH。因此，虽然比特币应用程序以易于阅读的方式呈现所有了80%以上的交易，但有时候会被偏离了常规的交易 难住。包含更复杂的锁定脚本，或不同SIGHASH标志，或多个输入和输出的交易显示了这些抽象的简单性和弱点。

每天都有数百个不包含P2PKH输出的交易在块上被确认。 blockchain浏览器经常向他们发出红色警告信息，表示无法解码地址。以下链接包含未完全解码的最新的“奇怪交易”：https：//blockchain.info/strange-transactions。

正如我们将在下一章中看到的，这些并不一定是奇怪的交易。它们是包含比常见的P2PKH更复杂的锁定脚本的交易。我们将学习如何解码和了解更复杂的脚本及其支持的应用程序。

7.1介绍

在上一章中，我们介绍了比特币交易的基本元素，并且查看了最常见的交易脚本类型，即P2PKH脚本。在本章中，我们将介绍更高级的脚本，以及如何使用它来构建具有复杂条件的交易。

首先，我们将看看多重签名脚本。接下来，我们将检查第二个最常见的交易脚本Pay-to-Script-Hash，它打开了一个复杂脚本的整个世界。然后，我们将检查新的脚本操作符，通过时间锁定将比特币添加时间维度。

7.2多重签名

多重签名脚本设置了一个条件，其中N个公钥被记录在脚本中，并且至少有M个必须提供签名来解锁资金。这也称为M-N方案，其中N是密钥的总数，M是验证所需的签名的数量。例如，2/3的多重签名是三个公钥被列为潜在签名人，至少有2个有效的签名才能花费资金。此时，标准多重签名脚本限制在最多15个列出的公钥，这意味着您可以从1到15之间的多重签名或该范围内的任何组合执行任何操作。在本书发布之前，限制15个已列出d的密钥可能会被解除，因此请检查isStandard（）函数以查看当前网络接受的内容。

设置M-N多重签名条件的锁定脚本的一般形式是：

```
M <Public Key 1> <Public Key 2> ... <Public Key N> N CHECKMULTISIG
```

M是花费输出所需的签名的数量，N是列出的公钥的总数。 设置2到3多重签名条件的锁定脚本如下所示：

```
2 <Public Key A> <Public Key B> <Public Key C> 3 CHECKMULTISIG
```

上述锁定脚本可由含有签名和公钥的脚本予以解锁： 或者由3个存档公钥中的任意2个相一致的私钥签名组合予以解锁。 两个脚本组合将形成一个验证脚本：

```
<Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3 CHECKMULT
ISIG
```

当执行时，只有当未解锁版脚本与解锁脚本设置条件相匹配时，组合脚本才显示得到结果为眞（Ture）。

上述例子中相应的设置条件即为：解锁脚本是否含有3个公钥中的任意2个相对应的私钥的有效签名。

CHECKMULTISIG执行中的bug

CHECKMULTISIG的执行中有一个bug，需要一些轻微的解决方法。 当CHECKMULTISIG执行时，它应该消耗堆栈(stack) 上的M + N + 2个项目作为参数。 然而，由于该错误，CHECKMULTISIG将弹出（pop）超出预期的额外值或一个值。 我们来看看这个更详细的/使用以前的/验证示例：

```
<Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3 CHECKMULT
ISIG
```

首先，CHECKMULTISIG弹出最上面的项目，这是N（在这个例子中N是“3”）。然后它弹出N个项目，这是可以签名的公钥。在这个例子中，公钥A，B和C.然后，它弹出一个项目，即M，仲裁（需要多少个签名）。这里M = 2。此时，CHECKMULTISIG应弹出最终的M个项目，这些是签名，并查看它们是否有效。

然而，不幸的是，实施中的错误导致CHECKMULTISIG再弹出一个项目（总共M + 1个）。检查签名时，不考虑额外的项目，因此它对CHECKMULTISIG本身没有直接影响。但是，必须存在额外的值，因为如果不存在，则当CHECKMULTISIG尝试弹出空堆栈时，会导致堆栈错误和脚本失败（将交易标记为无效）。因为额外的项目被忽略，它可以是任何东西，但通常使用0。

因为这个bug成为共识规则的一部分，所以现在它必须永远被复制。因此，正确的脚本验证将如下所示：

```
0 <Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3 CHECKMU
LTISIG
```

这样解锁脚本就不是下面的：

```
<Signature B> <Signature C>
```

而是：

```
0 <Signature B> <Signature C>
```

从现在开始，如果你看到一个multisig解锁脚本，你应该期望看到一个额外的0开始，其唯一的目的是解决一个bug，意外地成为一个共识规则的解决方法。 【译者注：即保证例子中有3个私钥签名（其中2有效签名，其中1个为0的无效签名）对应3个公钥用于检查多重签名，从而保证脚本不产生bug。】

7.3 P2SH（Pay-to-Script-Hash）

P2SH在2012年被作为一种新型、强大、且能大大简化复杂交易脚本的交易类型而引入。为进一步解释P2SH的必要性，让我们先看一个实际的例子。

在第1章中，我们曾介绍过Mohammed，一个迪拜的电子产品进口商。Mohammed的公司采用比特币多重签名作为其公司会计账簿记账要求。多重签名脚本是比特币高级脚本最为常见的运用之一，是一种具有相当大影响力的脚本。针对所有的顾客支付（即应收账款），Mohammed的公司要求采用多重签名交易。基于多重签名机制，顾客的任何支付都需要至少两个签名才能解锁，一个来自Mohammed，另一个来自其合伙人或拥有备份钥匙的代理人。这样的多重签名机制能为公司治理提供管控便利，同时也能有效防范盗窃、挪用和遗失。 最终的脚本非常长：

```
2 <Mohammed's Public Key> <Partner1 Public Key> <Partner2 Public Key> <Partner3 Public
 Key> <Attorney Public Key> 5 OP_C HECKMULTISIG
```

虽然多重签名十分强大，但其使用起来还是多有不便。基于之前的脚本，Mohammed必须在客户付款前将该脚本发送给每一位客户，而每一位顾客也必须使用特制的能产生客户交易脚本的比特币钱包软件，每位顾客还得学会如何利用脚本来完成交易。

此外，由于脚本可能包含特别长的公钥，最终的交易脚本可能是最初交易脚本长度的5倍之多。额外长度的脚本将给客户造成费用负担。最后，一个长的交易脚本将一直记录在所有节点的随机存储器的UTXO集中，直到该笔资金被使用。采用这种复杂输出脚本使得在实际交易中变得困难重重。

P2SH正是为了解决这一实际难题而被引入的，它旨在使复杂脚本的运用能与直接向比特币地址支付一样简单。在P2SH 支付中，复杂的锁定脚本被电子指纹所取代，电子指纹是指密码学中的哈希值。

当一笔交易试图支付UTXO时，要解锁支付脚本，它必须含有与哈希相匹配的脚本。P2SH的含义是，向与该哈希匹配的脚本支付，当输出被支付时，该脚本将在后续呈现。

在P2SH交易中，锁定脚本由哈希运算后的20字节的散列值取代，被称为赎回脚本。因为它在系统中是在赎回时出现而不是以锁定脚本模式出现。表7-1列示了非P2SH脚本，表7-2列示了P2SH脚本。

表7-1不含P2SH的复杂脚本

Locking Script	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 CHECKMULTISIG
Unlocking Script	Sig1 Sig2

表7-2 P2SH复杂脚本

Redeem Script	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 CHECKMULTISIG
Locking Script	HASH160 <20-byte hash of redeem script> EQUAL
Unlocking Script	Sig1 Sig2 <redeem script>

从表中可以看出，对于P2SH，详细描述了输出（赎回脚本）的条件的复杂脚本不会在锁定脚本中显示。相反，只有它的散列值在锁定脚本中呈现，并且兑换脚本本身之后作为解锁脚本在输出花费时的一部分出现。 这使得给矿工的交易费用从发送方转移到收款方，复杂的计算工作也从发送方转移到收款方。

[译者注]赎回脚本（Redeem Script）中的“2 Pubkey1 Pubkey2 Pubkey3 Pubkey4 Pubkey5 5 CHECKMULTISIG”的内容，没有出现在锁定脚本（Locking Script 表中第二行内容）中，但使用“2 Pubkey1 Pubkey2 Pubkey3 Pubkey4 Pubkey5 5 CHECKMULTISIG”（有520字节）进行哈希运算后的20字节的散列值取代之，然后将之（“2 Pubkey1 Pubkey2 Pubkey3 Pubkey4 Pubkey5 5 CHECKMULTISIG”）放到解锁脚本中（Unlocking Script）。这使得给矿工的交易费用从发送方转移到收款方，并且令复杂的计算工作也从从发送方转移到收款方。

让我们再看下Mohammed公司的例子，复杂的多重签名脚本和相应的P2SH脚本。 首先，Mohammed公司对所有顾客订单采用多重签名脚本：

2 <Mohammed's Public Key> 5 CHECKMULTISIG

如果占位符由实际的公钥（以04开头的520字节）替代，你将会看到的脚本会非常地长：

> 2
> 04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCC
> A01CDCA79DCC5C395D7EEC6984D83F1F50C900A24DD47F569FD4193AF5DE762
> C58704A2192968D8655D6A935BEAF2CA23E3FB87A3495E7AF308EDF08DAC3C1F
> CBFC2C75B4B0F4D0B1B70CD2423657738C0C2B1D5CE65C97D78D0E3422485800
> 8E8B49047E63248B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9
> D5737812F393DA7D4420D7E1A9162F0279CFC10F1E8E8F3020DECDBC3C0DD389
> D99779650421D65CBD7149B255382ED7F78E946580657EE6FDA162A187543A9D8
> 5BAAA93A4AB3A8F044DADA618D087227440645ABE8A35DA8C5B73997AD343BE5
> C2AFD94A5043752580AFA1ECED3C68D446BCAB69AC0BA7DF50D56231BE0AABF
> 1FDEEC78A6A45E394BA29A1EDF518C022DD618DA774D207D137AAB59E0B000E
> B7ED238F4D800 5 CHECKMULTISIG

整个脚本都可由仅为20个字节的密码哈希所取代，首先采用SH256哈希算法，随后对其运用RIPEMD160算法。20字节 的脚本为：

```
54c557e07dde5bb6cb791c7a540e0a4796f5e97
```

一笔P2SH交易运用锁定脚本将输出与哈希关联，而不是与前面特别长的脚本所关联。使用的锁定脚本为：

```
HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e EQUAL
```

正如你所看到的，这个脚本比前面的长脚本简短多了。取代“向该5个多重签名脚本支付”，这个P2SH等同于“向含该哈希的脚本支付”。顾客在向Mohammed公司支付时，只需在其支付指令中纳入这个非常简短的锁定脚本即可。当 Mohammed想要花费这笔UTXO时，附上原始赎回脚本（与UTXO锁定的哈希）和必要的解锁签名即可，如：

```
<Sig1> <Sig2> <2 PK1 PK2 PK3 PK4 PK5 5 CHECKMULTISIG>
```

两个脚本经由两步实现组合。 首先，将赎回脚本与锁定脚本比对以确认其与哈希是否匹配：

```
<2 PK1 PK2 PK3 PK4 PK5 5 CHECKMULTISIG> HASH160 <redeem scriptHash> EQUAL
```

假如赎回脚本与哈希匹配，解锁脚本会被执行以释放赎回脚本：

```
<Sig1> <Sig2> 2 PK1 PK2 PK3 PK4 PK5 5 CHECKMULTISIG
```

本章中描述的几乎所有脚本只能以P2SH脚本来实现。 它们不能直接用在UTXO的锁定脚本中。

7.3.1 P2SH地址

P2SH的另一重要特征是它能将脚本哈希编译为一个地址（其定义请见BIP0013 /BIP-13）。P2SH地址是基于Base58编码的一 个含有20个字节哈希的脚本，就像比特币地址是基于Base58编码的一个含有20个字节的公钥。由于P2SH地址采用5作为前缀，这导致基于Base58编码的地址以“3”开头。例如，Mohammed的脚本，基于Base58编码下的P2SH地址变为“39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw”。

此时，Mohammed可以将该地址发送给他的客户，这些客户可以 采用任何的比特币钱包实现简单支付，就像这是一个比特币地址一样。以“3”为前缀给予客户这是一种特殊类型的地址的暗示，该地址与一个脚本相对应而非与一个公钥相对应，但是它的效果与比特币地址支付别无二致。 P2SH地址隐藏了所有的复杂性，因此，运用其进行支付的人将不会看到脚本。

7.3.2 P2SH的优点

与直接使用复杂脚本以锁定输出的方式相比，P2SH具有以下特点：

- 在交易输出中，复杂脚本由简短电子指纹取代，使得交易代码变短。

- 脚本能被编译为地址，支付指令的发出者和支付者的比特币钱包不需要复杂工序就可以执行P2SH。
- P2SH将构建脚本的重担转移至接收方，而非发送方。
- P2SH将长脚本数据存储的负担从输出方（存储于UTXO集，影响内存）转移至输入方（存储在区块链里面）。
- P2SH将长脚本数据存储的重担从当前（支付时）转移至未来（花费时）。
- P2SH将长脚本的交易费成本从发送方转移至接收方，接收方在使用该笔资金时必须含有赎回脚本。

7.3.3赎回脚本和标准确认

在0.9.2版比特币核心客户端之前，P2SH仅限于标准比特币交易脚本类型（即通过标准函数检验的脚本）。这也意味着使用该笔资金的交易中的赎回脚本只能是标准化的P2PK、P2PKH或者多重签名，而非RETURN 和P2SH。

作为0.9.2版的比特币核心客户端，P2SH交易能包含任意有效的脚本，这使得P2SH标准更为灵活，也可以用于多种新的或复杂类型的交易。

请记住不能将P2SH植入P2SH赎回脚本，因为P2SH不能自循环。虽然在技术上可以将RETURN包含在赎回脚本中，但由于规则中没有策略阻止您执行此操作，因此在验证期间执行RETURN将导致交易被标记为无效，因此这是不实际的。

需要注意的是，因为赎回脚本只有在你试图发送一个P2SH输出时才会在比特币网络中出现，假如你将输出与一个无效的交易哈希锁定，则它将会被忽略。该UTXO将会被成功锁定，但是你将不能使用该笔资金，因为交易中含有赎回脚本，该脚本因是一个无效的脚本而不能被接受。这样的处理机制也衍生出一个风险，你可能将比特币锁定在一个未来不能被花费的P2SH中。因为比特币网络本身会接受这一P2SH，即便它与无效的赎回脚本所对应（因为该赎回脚本哈希没有对其所表征的脚本给出指令）。

注释 P2SH锁定脚本包含一个赎回脚本哈希，该脚本对于赎回脚本本身未提供任何描述。P2SH交易即便在赎回脚本无效的情况下也会被认为有效。如果处理不当，有可能会出现一个事故，即你的比特币可能会被锁死在P2SH这个交易中，导致你以后再也不能花费这笔比特币了。

7.4 数据记录输出（RETURN操作符）

比特币的去中心特点和时间戳账本机制，即区块链技术，其潜在运用将大大超越支付领域。许多开发者试图充分发挥交易脚本语言的安全性和可恢复性优势，将其运用于电子公证服务、证券认证和智能合约等领域。很多早期的开发者利用比特币这种能将交易数据放到区块链上的技术进行了很多尝试 ，例如，为文件记录电子指纹，则任何人都可以通过该机制在特定的日期建立关于文档存在性的证明。

运用比特币的区块链技术存储与比特币支付不相关数据的做法是一个有争议的话题。许多开发者认为其有滥用的嫌疑，因而试图予以阻止。另一些开发者则将之视为区块链技术强大功能的有力证明，从而试图给予大力支持。那些反对非支付相关应用的开发者认为这样做将引致“区块链膨胀”，因为所有的区块链节点都将以消耗磁盘存储空间为成本，负担存储此类 数据的任务。

更为严重的是，此类交易仅将比特币地址当作自由组合的20个字节而使用，进而会产生不能用于交易的UTXO。因为比特币地址只是被当作数据使用，并不与私钥相匹配，所以会导致UTXO不能被用于交易，因而是一种伪支付行为。因此，这些交易永远不会被花费，所以永远不会从UTXO集中删除，并导致UTXO数据库的大小永远增加或“膨胀”。

在0.9版的比特币核心客户端上，通过采用Return操作符最终实现了妥协。Return允许开发者在交易输出上增加80字节的非交易数据。然后，与伪交易型的UTXO不同，Return创造了一种明确的可复查的非交易型输出，此类数据无需存储于UTXO集。Return输出被记录在区块链上，它们会消耗磁盘空间，也会导致区块链规模的增加，但 它们不存储在UTXO集中，因此也不会使得UTXO内存膨胀，更不会以消耗代价高昂的内存为代价使全节点都不堪重负。
RETURN 脚本的样式：

```
RETURN <data>
```

“data”部分被限制为80字节，且多以哈希方式呈现，如32字节的SHA256算法输出。许多应用都在其前面加上前缀以辅助认定。例如，电子公正服务的证明材料采用8个字节的前缀“DOCPROOF”，在十六进制算法中，相应的ASCII码为 44 4f 43 50 52 4f 4f 46。

请记住 RETURN 不涉及可用于支付的解锁脚本的特点， RETURN 不能使用其输出中所锁定的资金，因此它也就没有必要记录在蕴含潜在成本的UTXO集中，所以 RETURN 实际是没有成本的。

RETURN 常为一个金额为0的比特币输出， 因为任何与该输出相对应的比特币都会永久消失。假如一笔 RETURN 被作为一笔交易的输入，脚本验证引擎将会阻止验证脚本的执行，将标记交易为无效。如果你碰巧将 RETURN 的输出作为另一笔交易的输入，则该交易是无效的。

一笔标准交易（通过了 isStandard() 函数检验的）只能有一个 RETURN 输出。但是单个 RETURN 输出能与任意类型的输出交易进行组合。

Bitcoin Core中添加了两个新版本的命令行选项。 选项datacarrier控制RETURN交易的中继和挖掘，默认设置为“1”以允许它们。 选项datacarriersize采用一个数字参数，指定RETURN脚本的最大大小（以字节为单位），默认为83字节，允许最多80个字节的RETURN数据加上一个字节的RETURN操作码和两个字节的PUSHDATA操作码。

注释 最初提出了RETURN，限制为80字节，但是当功能被释放时，限制被减少到40字节。2015年2月，在Bitcoin Core的0.10版本中，限制提高到80字节。 节点可以选择不中继或重新启动RETURN，或者只能中继和挖掘包含少于80字节数据的RETURN。

7.5时间锁（Timelocks）

时间锁是只允许在一段时间后才允许支出的交易。比特币从一开始就有一个交易级的时间锁定功能。它由交易中的nLocktime字段实现。在2015年底和2016年中期推出了两个新的时间锁定功能，提供UTXO级别的时间锁定功能。这些是CHECKLOCKTIMEVERIFY和CHECKSEQUENCEVERIFY。

时间锁对于后期交易和将资金锁定到将来的日期很有用。更重要的是，时间锁将比特币脚本扩展到时间的维度，为复杂的多级智能合同打开了大门。

7.5.1交易锁定时间（nLocktime）

比特币从一开始就有一个交易级的时间锁功能。交易锁定时间是交易级设置（交易数据结构中的一个字段），它定义交易有效的最早时间，并且可以在网络上中继或添加到区块链中。

锁定时间也称为nLocktime，是来自于Bitcoin Core代码库中使用的变量名称。在大多数交易中将其设置为零，以指示即时传播和执行。如果nLocktime不为零，低于5亿，则将其解释为块高度，这意味着交易无效，并且在指定的块高度之前未被中继或包含在块链中。

如果超过5亿，它被解释为Unix纪元时间戳（自Jan-1-1970之后的秒数），并且交易在指定时间之前无效。指定未来块或时间的nLocktime的交易必须由始发系统持有，并且只有在有效后才被发送到比特币网络。如果交易在指定的nLocktime之前传输到网络，那么第一个节点就会拒绝该交易，并且不会被中继到其他节点。使用nLocktime等同于一张延期支票。

7.5.1.1交易锁定时间限制

nLocktime就是一个限制，虽然它可以在将来花费，但是到现在为止，它并不能使用它们。我们来解释一下，下面的例子。

Alice签署了一笔交易，支付给Bob的地址，并将交易nLocktime设定为3个月。Alice把这笔交易发送给Bob。有了这个交易，Alice和Bob知道：

- 在3个月过去之前，Bob不能完成交易进行变现。
- Bob可以在3个月后接受交易。

  然而：

- Alice可以创建另一个交易，双重花费相同的输入，而不需要锁定时间。 因此，Alice可以在3个月过去之前花费相同的UTXO。

- Bob不能保证Alice不会这样做。

了解交易nLocktime的限制很重要。 唯一的保证是Bob在3个月过去之前无法兑换它。 不能保证Bob得到资金。 为了实现这样的保证，时间限制必须放在UTXO本身上，并成为锁定脚本的一部分，而不是交易。

这是通过下一种形式的时间锁定来实现的，称为检查锁定时间验证(CLTV)。

7.5.2检查锁定时间验证Check Lock Time Verify (CLTV)

2015年12月，引入了一种新形式的时间锁进行比特币软分叉升级。根据BIP-65中的规范，脚本语言添加了一个名为CHECKLOCKTIMEVERIFY（CLTV）的新脚本操作符。 CLTV是每个输出的时间锁定，而不是每个交易的时间锁定，与nLocktime的情况一样。这允许在应用时间锁的方式上具有更大的灵活性。 简单来说，通过在输出的赎回脚本中添加CLTV操作码来限制输出，从而只能在指定的时间过后使用。

注释 当nLocktime是交易级时间锁定时，CLTV是基于输出的时间锁。

CLTV不会取代nLocktime，而是限制特定的UTXO，并通过将nLocktim设置为更大或相等的值，从而达到在未来才能花费这笔钱的目的。

CLTV操作码采用一个参数作为输入，表示为与nLocktime（块高度或Unix纪元时间）相同格式的数字 。如VERIFY后缀所示，CLTV如果结果为FALSE，则停止执行脚本的操作码类型。如果结果为TRUE，则继续执行。

为了使用CLTV锁定输出，将其插入到创建输出的交易中的输出的赎回脚本中。例如，如果Alice支付Bob的地址，输出通常会包含一个这样的P2PKH脚本：

```
DUP HASH160 <Bob's Public Key Hash> EQUALVERIFY CHECKSIG
```

要锁定一段时间，比如说3个月以后，交易将是一个P2SH交易，其中包含一个赎回脚本：

```
<now + 3 months> CHECKLOCKTIMEVERIFY DROP DUP HASH160 <Bob's Public Key Hash> EQUALVER
IFY CHECKSIG
```

其中是从交易开始被挖矿时间起计3个月的块高度或时间值：当前块高度+12,960（块）或当前Unix纪元时间+7,760,000（秒）。现在，不要担心CHECKLOCKTIMEVERIFY之后的DROP操作码,下面很快就会解释。

当Bob尝试花费这个UTXO时，他构建了一个引用UTXO作为输入的交易。他使用他的签名和公钥在该输入的解锁脚本，并将交易nLocktime设置为等于或更大于Alice设置的CHECKLOCKTIMEVERIFY 时间锁。然后，Bob在比特币网络上广播交易。

Bob的交易评估如下。如果Alice设置的CHECKLOCKTIMEVERIFY参数小于或等于支出交易的nLocktime，脚本执行将继续（就好像执行“无操作”或NOP操作码一样）。否则，脚本执行停止，并且该交易被视为无效。 更确切地说，CHECKLOCKTIMEVERIFY失败并停止执行，标记交易无效（来源：BIP-65）：

1. 堆栈是空的要么
2. 堆栈中的顶部项小于0;要么
3. 顶层堆栈项和nLocktime字段的锁定时间类型（高度或者时间戳）不相同;要么
4. 顶层堆栈项大于交易的nLocktime字段;要么
5. 输入的nSequence字段为0xffffffff。

注释 CLTV和nLocktime使用相同的格式来描述时间锁定，无论是块高度还是自Unix纪元以秒钟以来所经过的时间。 最重要的是，在一起使用时，nLocktime的格式必须与输入中的CLTV格式相匹配，它们必须以秒为单位引用块高度或时间。

执行后，如果满足CLTV，则其之前的时间参数仍然作为堆栈中的顶级项，并且可能需要使用DROP进行删除，才能正确执行后续脚本操作码。 为此，您将经常在脚本中看到CHECKLOCKTIMEVERIFY＋DROP在一起使用。

通过将nLocktime与CLTV结合使用，交易锁定时间限制中描述的情况发生变化。 因为Alice锁定了UTXO本身，所以现在Bob或Alice在3个月的锁定时间到期之前不可能花费它。

通过将时间锁定功能直接引入到脚本语言中，CLTV允许我们开发一些非常有趣的复杂脚本。该标准在BIP-65（CHECKLOCKTIMEVERIFY）中定义（附录部分）。

7.5.3相对时间锁

nLocktime和CLTV都是绝对时间锁定，它们指定绝对时间点。接下来的两个时间锁定功能，我们将要考察的是相对时间锁定，因为它们将消耗输出的条件指定为从块链接中的输出确认起的经过时间。

相对时间锁是有用的，因为它们允许将两个或多个相互依赖的交易链接在一起，同时对依赖于从先前交易的确认所经过的时间的一个交易施加时间约束。换句话说，在UTXO被记录在块状块之前，时钟不开始计数。这个功能在双向状态通道和闪电网络中特别有用，我们将在后面章节[state_channels]中看到。

相对时间锁，如绝对时间锁定，同时具有交易级功能和脚本级操作码。交易级相对时间锁定是作为对每个交易输入中设置的交易字段nSequence的值的共识规则实现的。脚本级相对时间锁定使用CHECKSEQUENCEVERIFY（CSV）操作码实现。

相对时间锁是根据BIP-68与BIP－112的规范共同实现的，其中BIP-68通过与相对时间锁运用一致性增强的数字序列实现，BIP-112中是运用到了CHECKSEQUENCEVERIFY这个操作码实现。

BIP-68和BIP-112是在2016年5月作为软分叉升级时被激活的一个共识规则。

7.5.4 nSequence相对时间锁

相对时间锁定可以在每个输入中设置好，其方法是在每个输入中加多一个nSequence字段。

7.5.4.1 nSequence的本义

nnSequence字段的设计初心是想让交易能在在内存中修改，可惜后面从未运用过，使用nSequence这个字段时，如果输入的交易的序列值小于2^32 （0xFFFFFFFF），就表示尚未“确定”的交易。

这样的交易将在内存池中保存，直到被另一个交易消耗相同输入并具有较大nSequence值的代替。一旦收到一个交易，其投入的nSequence值为2^32，那么它将被视为“最终确定”并开采。 nSequence的原始含义从未被正确实现，并且在不利用时间锁定的交易中nSequence的值通常设置为2^32。对于具有nLocktime或CHECKLOCKTIMEVERIFY的交易，nSequence值必须设置为小于2^32，以使时间锁定器有效。通常设置为2^32 - 1（0xFFFFFFFE）。

7.5.4.2 nSequence作为一个共同执行的相对时间锁定

由于BIP-68的激活，新的共识规则适用于任何包含nSequence值小于2^31的输入的交易（bit 1<<31 is not set）。以编程方式，这意味着如果没有设置最高有效（bit 1<<31），它是一个表示“相对锁定时间”的标志。否则（bit 1<<31set），nSequence值被保留用于其他用途，例如启用CHECKLOCKTIMEVERIFY，nLocktime，Opt-In-Replace-By-Fee以及其他未来的新产品。

一笔输入交易，当输入脚本中的nSequence值小于2^31时，就是相对时间锁定的输入交易。这种交易只有到了相对锁定时间后才生效。例如，具有30个区块的nSequence相对时间锁的一个输入的交易只有在从输入中引用的UTXO开始的时间起至少有30个块时才有效。由于nSequence是每个输入字段，因此交易可能包含任何数量的时间锁定输入，所有这些都必须具有足够的时间以使交易有效。

交易可以包括时间锁定输入（nSequence <2^31）和没有相对时间锁定（nsequence> = 2^31）的输入。 nSequence值以块或秒为单位指定，但与nLocktime中使用的格式略有不同。类型标志用于区分计数块和计数时间（以秒为单位）的值。类型标志设置在第23个最低有效位（即值1 << 22）。如果设置了类型标志，则nSequence值将被解释为512秒的倍数。如果未设置类型标志，则nSequence值被解释为块数。

当将nSequence解释为相对时间锁定时，只考虑16个最低有效位。一旦评估了标志（位32和23），nSequence值通常用16位掩码（例如nSequence＆0x0000FFFF）“屏蔽”。 下图显示由BIP-68定义的nSequence值的二进制布局。

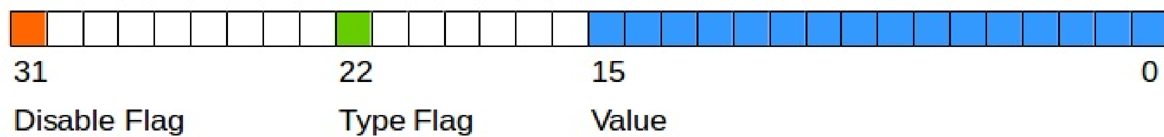


Figure 1. BIP-68 definition of nSequence encoding (Source: BIP-68)

基于nSequence值的一致执行的相对时间锁定在BIP-68中。标准定义在BIP-68, Relative lock-time using consensus-enforced sequence numbers.

7.5.5 带CSV的相对时间锁

就像CLTV和nLocktime一样，有一个脚本操作码用于相对时间锁定，它利用脚本中的nSequence值。该操作码是CHECKSEQUENCEVERIFY，通常简称为CSV。 在UTXO的赎回脚本中评估时，CSV操作码仅允许在输入nSequence值大于或等于CSV参数的交易中进行消耗。实质上，这限制了UTXO的消耗，直到UTXO开采时间过了一定数量的块或秒。

与CLTV一样，CSV中的值必须与相应nSequence值中的格式相匹配。如果CSV是根据块指定的，那么nSequence也是如此。如果以秒为单位指定CSV，那么nSequence也是如此。

当几个（已经形成链）交易被保留为"脱链"时，创建和签名这几个（已经形成链）交易但不传播时，CSV的相对时间锁特别有用。在父交易已被传播，直到消耗完相对锁定时间，才能使用子交易。这个用例的一个应用可以在state_channels和lightning_network/请加上链接 章节中看到。 CSV 细节参见 BIP-112, CHECKSEQUENCEVERIFY.

7.5.6中位时间过去Median-Time-Past

作为激活相对时间锁定的一部分，时间锁定（绝对和相对）的"时间"方式也发生了变化。在比特币中，墙上时间（wall time）和共识时间之间存在微妙但非常显著的差异。比特币是一个分散的网络，这意味着每个参与者都有自己的时间观。网络上的事件不会随时随地发生。网络延迟必须考虑到每个节点的角度。最终，所有内容都被同步，以创建一个共同的分类帐。比特币在过去存在的分类账状态中每10分钟达成一个新的共识。

区块头中设置的时间戳由矿工设定。共识规则允许一定的误差来解决分散节点之间时钟精度的问题。然而，这诱惑了矿工去说谎，以便通过包括还不在范围内的时间交易来赚取额外矿工费。有关详细信息，请参阅以下部分。

为了杜绝矿工说谎，加强时间安全性，在相对时间锁的基础上又新增了一个BIP。这是BIP-113，它定义了一个称为"中位时间过去 /（Median-Time-Past）"的新的共识测量机制。通过取最后11个块的时间戳并计算其中位数作为"中位时间过去"的值。这个中间时间值就变成了共

识时间，并被用于所有的时间计算。过去约两个小时的中间点，任何一个块的时间戳的影响减小了。通过这个方法，没有一个矿工可以利用时间戳从具有尚未成熟的时间段的交易中获取非法矿工费。

Median-Time-Past更改了nLocktime，CLTV，nSequence和CSV的时间计算的实现。由Median-Time-Past计算的共识时间总是大约在挂钟时间后一个小时。如果创建时间锁交易，那么要在nLocktime，nSequence，CLTV和CSV中进行编码的估计所需值时，应该考虑它。Median-Time-Past细节参见BIP-113.

7.5.7针对费用狙击（Fee Sniping）的时间锁定

费用狙击是一种理论攻击情形，矿工试图从将来的块（挑选手续费较高的交易）重写过去的块，实现“狙击”更高费用的交易，以最大限度地提高盈利能力。

例如，假设存在的最高块是块＃100,000。如果不是试图把＃100,001号的矿区扩大到区块链，那么一些矿工们会试图重新挖矿＃100,000。这些矿工可以选择在候选块＃100,000中包括任何有效的交易（尚未开采）。他们不必使用相同的交易来恢复块。事实上，他们有动力选择最有利可图（最高每kBB）的交易来包含在其中。它们可以包括处于“旧”块＃100,000中的任何交易，以及来自当前内存池的任何交易。当他们重新创建块＃100,000时，他们本质上可以将交易从“现在”提取到重写的“过去”中。

今天，这种袭击并不是非常有利可图，因为回报奖励（因为包括一定数量的比特币奖励）远远高于每个区块的总费用。但在未来的某个时候，交易费将是奖励的大部分（甚至是奖励的整体）。那时候这种情况变得不可避免了。

为了防止“费用狙击”，当Bitcoin Core /钱包 创建交易时，默认情况下，它使用nLocktime将它们限制为“下一个块”。在我们的环境中，Bitcoin Core /钱包将在任何创建的交易上将nLocktime设置为100,001。在正常情况下，这个nLocktime没有任何效果 - 交易只能包含在＃100,001块中，这是下一个区块。 但是在区块链分叉攻击的情况下，由于所有这些交易都将被时间锁阻止在＃100,001，所以矿工们无法从筹码中提取高额交易。他们只能在当时有效的任何交易中重新挖矿#100,000，这导致实质上不会获得新的费用。 为了实现这一点，Bitcoin Core/钱包将所有新交易的nLocktime设置为，并将所有输入上的nSequence设置为0xFFFFFFFE以启用nLocktime。

7.6具有流量控制的脚本（条件子句 (Conditional Clauses)）

比特币脚本的一个更强大的功能是流量控制，也称为条件条款。您可能熟悉使用构造IF ... THEN ... ELSE的各种编程语言中的流控制。比特币条件条款看起来有点不同，但是基本上是相同的结构。

在基本层面上，比特币条件操作码允许我们构建一个具有两种解锁方式的赎回脚本，这取决于评估逻辑条件的TRUE / FALSE结果。例如，如果x为TRUE，则赎回脚本为A，ELSE赎回脚本为B. 此外，比特币条件表达式可以无限期地“嵌套”，这意味着这个条件语句可以包含其中的另外一个条件，另外一个条件其中包含别的条件等等 。Bitcoin脚本流控制可用于构造非常复杂的脚本，具有数百甚至数千个可能的执行路径。嵌套没有限制，但协商一致的规则对脚本的最大大小（以字节为单位）施加限制。

比特币使用IF，ELSE，ENDIF和NOTIF操作码实现流量控制。此外，条件表达式可以包含布尔运算符，如BOOLAND，BOOLOR和NOT。

乍看之下，您可能会发现比特币的流量控制脚本令人困惑。那是因为比特币脚本是一种堆栈语言。同样的方式，当1+1看起来“向后”当表示为1 1 ADD时，比特币中的流控制条款也看起来“向后”（backward）。 在大多数传统（程序）编程语言中，流控制如下所示： 大多数编程语言中的流控制伪代码

```
if (condition):
  code to run when condition is true
else:
  code to run when condition is false
code to run in either case
```

在基于堆栈的语言中，比如比特币脚本，逻辑条件出现在IF之前，这使得它看起来像“向后”，如下所示： Bitcoin脚本流控制

```
 condition
IF
  code to run when condition is true
ELSE
  code to run when condition is false
ENDIF
code to run in either case
```

阅读Bitcoin脚本时，请记住，评估的条件是在IF操作码之前。

7.6.1带有VERIFY操作码的条件子句

比特币脚本中的另一种条件是任何以VERIFY结尾的操作码。 VERIFY后缀表示如果评估的条件不为TRUE，脚本的执行将立即终止，并且该交易被视为无效。 与提供替代执行路径的IF子句不同，VERIFY后缀充当保护子句，只有在满足前提条件的情况下才会继续。

例如，以下脚本需要Bob的签名和产生特定哈希的前图像（秘密地）。

解锁时必须满足这两个条件：

1)具有EQUALVERIFY保护子句的赎回脚本。

```
HASH160 <expected hash> EQUALVERIFY <Bob's Pubkey> CHECKSIG
```

为了兑现这一点，Bob必须构建一个解锁脚本，提供有效的前图像和签名：

2)一个解锁脚本以满足上述赎回脚本。

```
<Bob's Sig> <hash pre-image>
```

没有前图像，Bob无法访问检查其签名的脚本部分。

该脚本可以用IF编写： 具有IF保护条款的兑换脚本

```
HASH160 <expected hash> EQUAL
IF
   <Bob's Pubkey> CHECKSIG
ENDIF
```

Bob的解锁脚本是一样的： 解锁脚本以满足上述兑换脚本

```
<Bob's Sig> <hash pre-image>
```

使用IF的脚本与使用具有VERIFY后缀的操作码相同; 他们都作为保护条款。 然而，VERIFY的构造更有效率，使用较少的操作码。

那么，我们什么时候使用VERIFY，什么时候使用IF？ 如果我们想要做的是附加一个前提条件（保护条款），那么验证是更好的。 然而，如果我们想要有多个执行路径（流控制），那么我们需要一个IF ... ELSE流控制子句。

提示 诸如EQUAL之类的操作码会将结果（TRUE / FALSE）推送到堆栈上，留下它用于后续操作码的评估。 相比之下，操作码EQUALVERIFY后缀不会在堆栈上留下任何东西。 在VERIFY中结束的操作码不会将结果留在堆栈上。

7.6.2在脚本中使用流控制

比特币脚本中流量控制的一个非常常见的用途是构建一个提供多个执行路径的赎回脚本，每个脚本都有一种不同的赎回UTXO的方式。

我们来看一个简单的例子，我们有两个签名人，Alice和Bob，两人中任何一个都可以兑换。 使用多重签名，这将被表示为1-of-2 多重签名脚本。 为了示范，我们将使用IF子句做同样的事情：

```
IF
 <Alice's Pubkey> CHECKSIG
ELSE
 <Bob's Pubkey> CHECKSIG
ENDIF
```

看这个赎回脚本，你可能会想："条件在哪里？"IF子句之前没有什么！" 条件不是赎回脚本的一部分。

相反，该解锁脚本将提供该条件，允许Alice和Bob"选择"他们想要的执行路径。

Alice用解锁脚本兑换了这个：

```
<Alice's Sig> 1
```

最后的1作为条件（TRUE），将使IF子句执行Alice具有签名的第一个兑换路径。

为了兑换这个Bob，他必须通过给IF子句赋一个FALSE值来选择第二个执行路径：

```
<Bob's Sig> 0
```

Bob的解锁脚本在堆栈中放置一个0，导致IF子句执行第二个（ELSE）脚本，这需要Bob的签名。

由于可以嵌套IF子句，所以我们可以创建一个"迷宫"的执行路径。 解锁脚本可以提供一个选择执行路径实际执行的"地图"：

```
IF
    script A
ELSE
   IF
script B
  ELSE
script C
  ENDIF
ENDIF
```

在这种情况下，有三个执行路径（脚本A，脚本B和脚本C）。 解锁脚本以TRUE或FALSE值的形式提供路径。

要选择路径脚本B，例如，解锁脚本必须以1 0（TRUE，FALSE）结束。

这些值将被推送到堆栈，以便第二个值（FALSE）结束于堆栈的顶部。 外部IF子句弹出FALSE值并执行第一个ELSE子句。 然后，TRUE值移动到堆栈的顶部，并通过内部（嵌套）IF来评估，选择B执行路径。

使用这个结构，我们可以用数十或数百个执行路径构建赎回脚本，每个脚本提供了一种不同的方式来兑换UTXO。 要花费，我们构建一个解锁脚本，通过在每个流量控制点的堆栈上放置相应的TRUE和FALSE值来导航执行路径。

7.7复杂的脚本示例

在本节中，我们将本章中的许多概念合并成一个例子。 我们的例子使用了迪拜公司所有者Mohammed的故事，他们正在经营进出口业务。

在这个例子中，Mohammed希望用灵活的规则建立公司资本账户。他创建的方案需要不同级别的授权，具体取决于时间锁定。

多重签名的计划的参与者是Mohammed，他的两个合作伙伴Saeed和Zaira，以及他们的公司律师Abdul。三个合作伙伴根据多数规则作出决定，因此三者中的两个必须同意。然而，如果他们的钥匙有问题，他们希望他们的律师能够用三个合作伙伴签名之一收回资金。最后，如果所有的合作伙伴一段时间都不可用或无行为能力，他们希望律师能够直接管理该帐户。

这是Mohammed设计的脚本： 具有时间锁定（Timelock）变量的多重签名

```
IF
  IF
    2
  ELSE
    <30 days> CHECKSEQUENCEVERIFY DROP
    <Abdul the Lawyer's Pubkey> CHECKSIGVERIFY
    1
  ENDIF
  <Mohammed's Pubkey> <Saeed's Pubkey> <Zaira's Pubkey> 3 CHECKMULTISIG
ELSE
  <90 days> CHECKSEQUENCEVERIFY DROP
  <Abdul the Lawyer's Pubkey> CHECKSIG
ENDIF
```

Mohammed的脚本使用嵌套的IF ... ELSE流控制子句来实现三个执行路径。

在第一个执行路径中，该脚本作为三个合作伙伴的简单的2-of-3 multisig操作。

该执行路径由第3行和第9行组成。第3行将multisig的定额设置为2（2 - 3）。

该执行路径可以通过在解锁脚本的末尾设置TRUE TRUE来选择： 解锁第一个执行路径的脚本（2-of-3 multisig）

```
0 <Mohammed's Sig> <Zaira's Sig> TRUE TRUE
```

提示 此解锁脚本开头的0是因为CHECKMULTISIG中的错误从堆栈中弹出一个额外的值。 额外的值被CHECKMULTISIG忽略，否则脚本签名将失败。 推送0（通常）是解决bug的方法，如CHECKMULTISIG执行中的错误章节所述。

第二个执行路径只能在UTXO创建30天后才能使用。 那时候，它需要签署Abdul（律师）和三个合作伙伴之一（三分之一）。

这是通过第7行实现的，该行将多选的法定人数设置为1。要选择此执行路径，解锁脚本将以FALSE TRUE结束： 解锁第二个执行路径的脚本(Lawyer + 1-of-3）

```
0 <Saeed's Sig> <Abdul's Sig> FALSE TRUE
```

提示 为什么先FALSE后TRUE？ 反了吗？这是因为这两个值被推到堆栈，所以先push FALSE，然后push TRUE。 因此，第一个IF操作码首先弹出的是TRUE。

最后，第三个执行路径允许律师单独花费资金，但只能在90天之后。 要选择此执行路径，解锁脚本必须以FALSE结束： 解锁第三个执行路径的脚本（仅适用于律师）

```
<Abdul's Sig> FALSE
```

在纸上运行脚本来查看它在堆栈(stack)上的行为。

阅读这个例子还需要考虑几件事情。 看看你能找到答案吗？

- 为什么律师可以随时通过在解锁脚本中选择FALSE来兑换第三个执行路径？
- 在UTXO开采后分别有多少个执行路径可以使用5,35与105天？
- 如果律师失去钥匙，资金是否流失？ 如果91天过去了，你的答案是否会改变？
- 合作伙伴如何每隔29天或89天“重置”一次，以防止律师获得资金？
- 为什么这个脚本中的一些CHECKSIG操作码有VERIFY后缀，而其他的没有？

8.1 P2P网络架构

比特币采用了基于国际互联网（Internet）的P2P（peer-to-peer）网络架构。P2P是指位于同一网络中的每台计算机都彼此对等，各个节点共同提供网络服务，不存在任何“特殊”节点。每个网络节点以“扁平（flat）”的拓扑结构相互连通。 在P2P网络中不存在任何服务端（server）、中央化的服务、以及层级结构。P2P网络的节点之间交互运作、协同处理：每个节点在对外提供服务的同时也使用网络中其他节点所提供的服务。P2P网络也因此具有可靠性、去中心化，以 及开放性。早期的国际互联网就是P2P网络架构的一个典型用例：IP网络中的各个节点完全平等。当今的互联网架构具有分层架构，但是IP协议仍然保留了扁平拓扑的结构。在比特币之外，规模最大也最成功的P2P技术应用是在文件分享 领域：Napster是该领域的先锋，BitTorrent是其架构的最新演变。

比特币所采用的P2P网络架构不仅仅是选择拓扑结构这样简单。比特币被设计为一种点对点的数字现金系统，它的网络架构既是这种核心特性的反映，也是该特性的基石。去中心化控制是设计时的核心原则，它只能通过维持一种扁平化、 去中心化的P2P共识网络来实现。

“比特币网络”是按照比特币P2P协议运行的一系列节点的集合。除了比特币P2P协议之外，比特币网络中也包含其他协议。例如Stratum协议就被应用于挖矿、以及轻量级或移动端比特币钱包之中。网关（gateway）路由服务器提供这些协 议，使用比特币P2P协议接入比特币网络，并把网络拓展到运行其他协议的各个节点。例如，Stratum服务器通过 Stratum协议将所有的Stratum挖矿节点连接至比特币主网络、并将Stratum协议桥接（bridge）至比特币P2P协议之 上。我们使用“扩展比特币网络（extended bitcoin network）”指代所有包含比特币P2P协议、矿池挖矿协议、Stratum 协议以及其他连接比特币系统组件相关协议的整体网络结构。

8.2 节点类型及角色

尽管比特币P2P网络中的各个节点相互对等，但是根据所提供的功能不同，各节点可能具有不同的角色。每个比特币节点都是路由、区块链数据库、挖矿、钱包服务的功能集合。一个全节点（full node）包括如图8-1所示的四个功能：

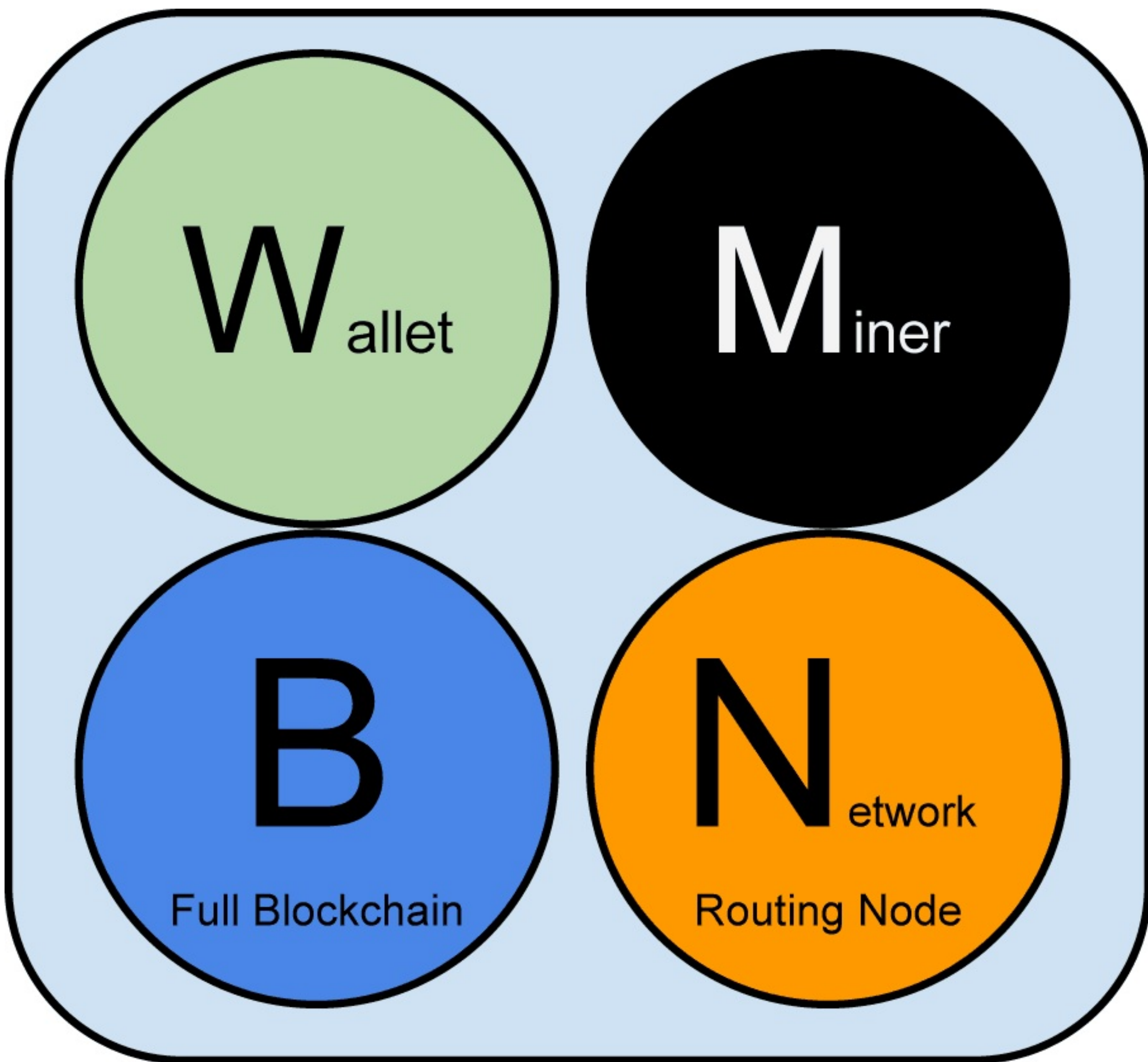


图8-1比特币网络节点，具有所有四个功能：钱包，矿工，完整的区块链数据库和网络路由

每个节点都参与全网络的路由功能，同时也可能包含其他功能。每个节点都参与验证并传播交易及区块信息，发现并维持与对等节点的连接。在图8-1所示的全节点用例中，名为“网络路由节点”的橙色圆圈字母‘N’即表示该路由功能。

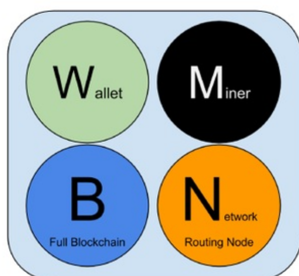
一些节点保有一份完整的、最新的区块链拷贝，这样的节点被称为“全节点”。全节点能够独立自主地校验所有交易，而不需借由任何外部参照。另外还有一些节点只保留了区块链的一部分，它们通过一种名为“简易支付验证（SPV）”的方 式来完成交易验证。这样的节点被称为“SPV节点”，又叫“轻量级节点”。在如上图所示的全节点用例中，名为完整区块链的蓝色圆圈字母“B”即表示了全节点区块链数据库功能。在图8-3中，SPV节点没有此蓝色圆圈，以示它们没有区块链的完整拷贝。

挖矿节点通过运行在特殊硬件设备上的工作量证明（proof-of-work）算法，以相互竞争的方式创建新的区块。一些挖矿 节点同时也是全节点，保有区块链的完整拷贝；还有一些参与矿池挖矿的节点是轻量级节点，它们必须依赖矿池服务器维护的全节点进行工作。在全节点用例中，挖矿功能如图中名为“矿工”的黑色圆圈字母“M”所示。

用户钱包也可以作为全节点的一部分，这在桌面比特币客户端中比较常见。当前，越来越多的用户钱包都是SPV节点， 尤其是运行于诸如智能手机等资源受限设备上的比特币钱包应用；而这正变得越来越普遍。在图8-1中，名为“钱包”的绿色圆圈字母“W”代表钱包功能。

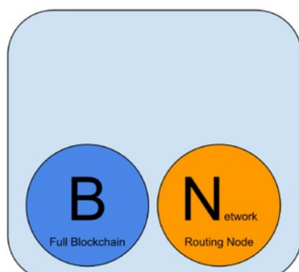
在比特币P2P协议中，除了这些主要的节点类型之外，还有一些服务器及节点也在运行着其他协议，例如特殊矿池挖矿 协议、轻量级客户端访问协议等。

图8-2描述了扩展比特币网络中最为常见的节点类型。



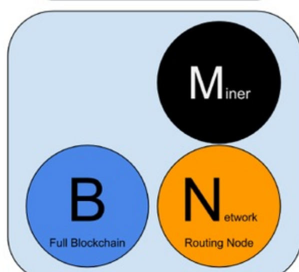
Reference Client (Bitcoin Core)

Contains a Wallet, Miner, full Blockchain database, and Network routing node on the bitcoin P2P network.



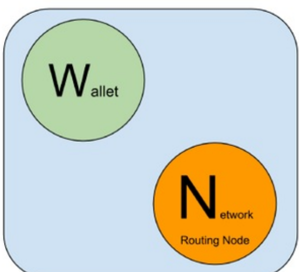
Full Block Chain Node

Contains a full Blockchain database, and Network routing node on the bitcoin P2P network.



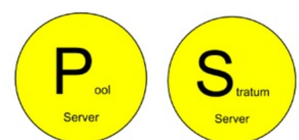
Solo Miner

Contains a mining function with a full copy of the blockchain and a bitcoin P2P network routing node.



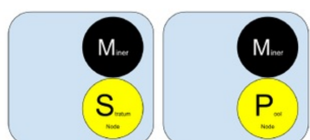
Lightweight (SPV) wallet

Contains a Wallet and a Network node on the bitcoin P2P protocol, without a blockchain.



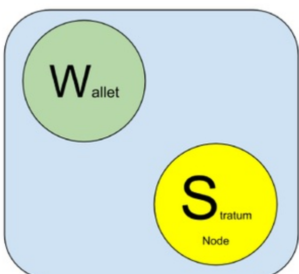
Pool Protocol Servers

Gateway routers connecting the bitcoin P2P network to nodes running other protocols such as pool mining nodes or Stratum nodes.



Mining Nodes

Contain a mining function, without a blockchain, with the Stratum protocol node (S) or other pool (P) mining protocol node.



Lightweight (SPV) Stratum wallet

Contains a Wallet and a Network node on the Stratum protocol, without a blockchain.

图8-2描述了扩展比特币网络中最为常见的节点类型。

8.3 扩展比特币网络

运行比特币P2P协议的比特币主网络由大约5000-8000个运行着不同版本比特币核心客户端（Bitcoin Core）的监听节 点、以及几百个运行着各类比特币P2P协议的应用（例如Bitcoin Classic, Bitcoin Unlimited, BitcoinJ, Libbitcoin, btcd, and bcoin等）的节点组成。比特币P2P网络中的一小部分节点也是挖矿节点，它们竞争挖矿、验证交易、并创建新的区块。许多连接到比特币网络的大型公司运行 着基于Bitcoin核心客户端的全节点客户端，它们具有区块链的完整拷贝及网络节点，但不具备挖矿及钱包功能。这些节点是网络中的边缘路由器（edge routers），通过它们可以搭建其他服务，例如交易所、钱包、区块浏览器、商家支付处理（merchant payment processing）等。

如前文所述，扩展比特币网络既包括了运行比特币P2P协议的网络，又包含运行特殊协议的网络节点。比特币P2P主网络上连接着许多矿池服务器以及协议网关，它们把运行其他协议的节点连接起来。这些节点通常都是矿池挖矿节点（参见挖矿章节）以及轻量级钱包客户端，它们通常不具备区块链的完整备份。

图8-3描述了扩展比特币网络，它包括了多种类型的节点、网关服务器、边缘路由器、钱包客户端以及它们相互连接所 需的各类协议。

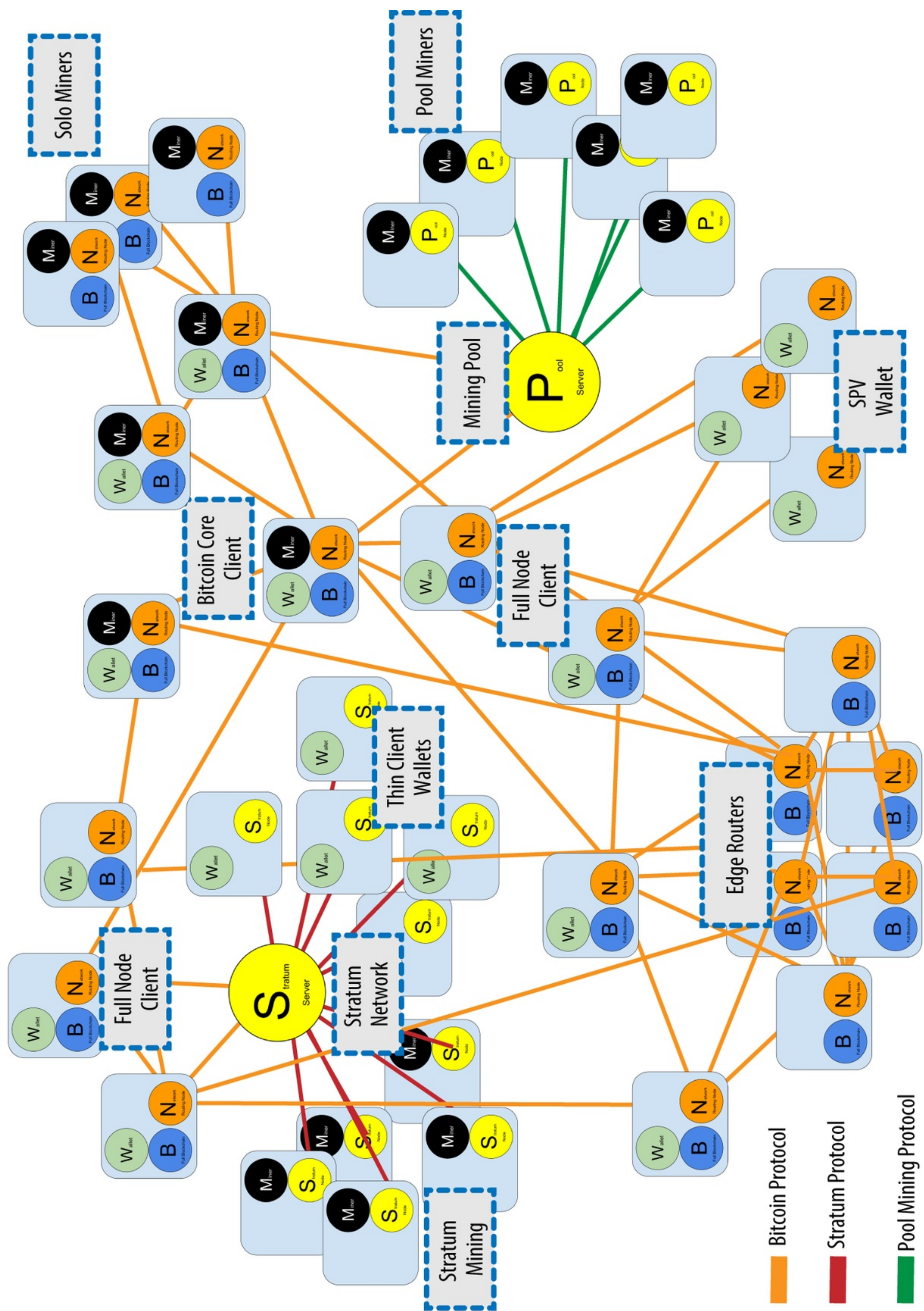


图8-3 显示各种节点类型，网关和协议的扩展比特币网络

8.4比特币传播网络

虽然比特币P2P网络服务于各种各样的节点类型的一般需求，但是对于比特币挖掘节点的专门需求，它显示出太高的网络延迟。

比特币矿业公司正在进行时间敏感的竞争，以解决工作证明问题，并扩大区块链（参见[挖矿]章节）。在参加比赛时，比特币矿工必须最大限度地缩短获胜块的传播与下一轮比赛开始之间的时间。在采矿方面，网络延迟与利润率直接相关。

比特币传播网络是一种尝试最小化矿工之间传输块的延迟的网络。原始的比特币传播网络是由核心开发商Matt Corallo于2015年创建的，以便能够以非常低的延迟在矿工之间快速同步块。该网络由世界各地的亚马逊Web服务基础架构上托管的几个专门的节点组成，并且连接大多数矿工和采矿池。

原始的比特币传播网络在2016年被替换为*Fast Internet Bitcoin Relay Engine* or *FIBRE*，也由核心开发商Matt Corallo创建。 FIBER是一种基于UDP的中继网络，可以中继节点网络内的块。 FIBER实现了compact block，以进一步减少传输的数据量和网络延迟。

康奈尔大学研究的另一个中继网络（仍在提案阶段）是 *Falcon*。 Falcon使用“直通路由”而不是“存储转发”来减少延迟，通过传播块的部分，而不是等待直到接收到完整的块。

传播网络不是比特币的P2P网络的替代品。相反，它们是覆盖网络，在具有特殊需求的节点之间提供额外的连接像高速公路不是农村道路的替代品，而是交通繁忙的两点之间的快捷方式，您仍然需要小路连接高速公路。

8.5 网络发现

当新的网络节点启动后，为了能够参与协同运作，它必须发现网络中的其他比特币节点。新的网络节点必须发现至少一个网络中存在的节点并建立连接。由于比特币网络的拓扑结构并不基于节点间的地理位置，因此各个节点之间的地理信息完全无关。在新节点连接时，可以随机选择网络中存在的比特币节点与之相连。

节点通常采用TCP协议、使用8333端口（该端口号通常是比特币所使用的，除8333端口外也可以指定使用其他端口） 与已知的对等节点建立连接。在建立连接时，该节点会通过发送一条包含基本认证内容的version消息开始“握手”通信过 程(见图8-4)。这一过程包括如下内容：

▷ nVersion

定义了客户端所“说出”的比特币P2P协议所采用的版本（例如：70002）。

▷ nLocalServices

一组该节点支持的本地服务列表，当前仅支持NODE_NETWORK

▷ nTime

当前时间

▷ addrYou

当前节点可见的远程节点的IP地址

▷ addrMe

本地节点所发现的本机IP地址

▷ subver

指示当前节点运行的软件类型的子版本号（例如："/Satoshi:0.9.2.1/"）

▷ BaseHeight

当前节点区块链的区块高度 （version网络消息的具体用例请参见GitHub ）

版本消息始终是任何对等体发送给另一个对等体的第一条消息。 接收版本消息的本地对等体将检查远程对等体报告的nVersion，并确定远端对等体是否兼容。 如果远程对等体兼容，则本地对等体将确认版本消息，并通过发送一个verack建立连接。

新节点如何找到对等体？ 第一种方法是使用多个“DNS种子”来查询DNS，这些DNS服务器提供比特币节点的IP地址列表。 其中一些DNS种子提供了稳定的比特币侦听节点的静态IP地址列表。 一些DNS种子是BIND（Berkeley Internet Name Daemon）的自定义实现，它从搜索器或长时间运行的比特币节点收集的比特币节点地址列表中返回一个随机子集。 Bitcoin Core客户端包含五种不同DNS种子的名称。 不同DNS种子的所有权和多样性的多样性为初始引导过程提供了高水平的可靠性。 在Bitcoin Core客户端中，使用DNS种子的选项由选项switch -dnsseed控制（默认设置为1，以使用DNS种子）。

或者，不知道网络的引导节点必须被给予至少一个比特币节点的IP地址，之后可以通过进一步介绍来建立连接。 命令行参数-seednode可用于连接到一个节点，仅用于将其用作种子。 在使用初始种子节点形成介绍后，客户端将断开连接并使用新发现的对等体。

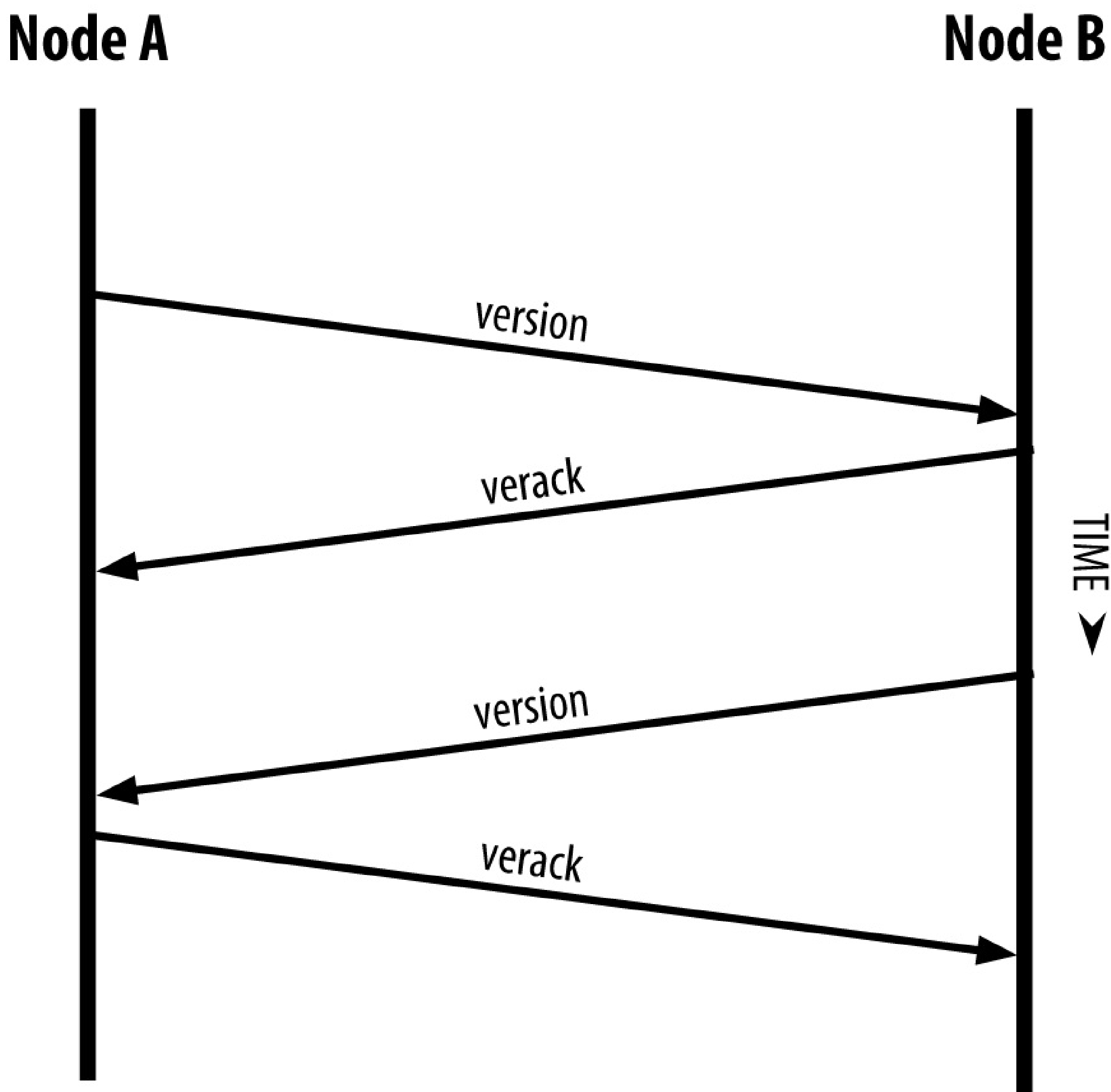


图8-4 对等体之间的初始握手

当建立一个或多个连接后，新节点将一条包含自身IP地址的addr消息发送给其相邻节点。相邻节点再将此条addr消息依 次转发给它们各自的相邻节点，从而保证新节点信息被多个节点所接收、保证连接更稳定。另外，新接入的节点可以向 它的相邻节点发送getaddr消息，要求它们返回其已知对等节点的IP地址列表。通过这种方式，节点可以找到需连接到 的对等节点，并向网络发布它的消息以便其他节点查找。图8-5描述了这种地址发现协议。

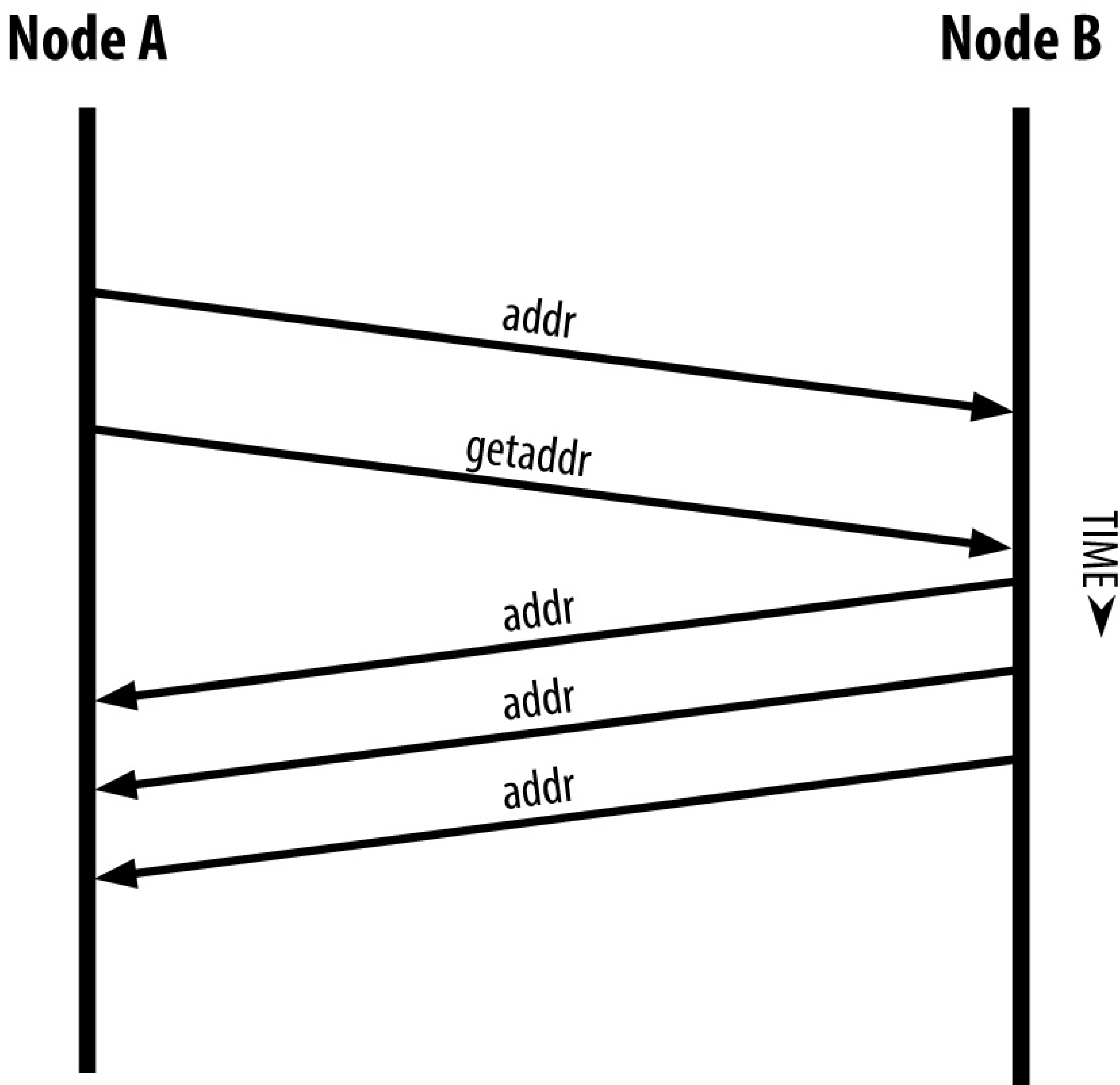


图8-5 地址传播和发现

节点必须连接到若干不同的对等节点才能在比特币网络中建立通向比特币网络的种类各异的路径（path）。由于节点可以随时加入和离开，通讯路径是不可靠的。因此，节点必须持续进行两项工作：在失去已有连接时发现新节点，并在其他节点启动时为其提供帮助。节点启动时只需要一个连接，因为第一个节点可以将它引荐给它的对等节点，而这些节点又会进一步提供引荐。一个节点，如果连接到大量的其他对等节点，这既没必要，也是对网络资源的浪费。在启动完成 后，节点会记住它最近成功连接的对等节点；因此，当重新启动后它可以迅速与先前的对等节点网络重新建立连接。如果先前的网络的对等节点对连接请求无应答，该节点可以使用种子节点进行重启动。

在运行比特币核心客户端的节点上，您可以使用 getpeerinfo 命令列出对等节点连接信息：

```
$ bitcoin-cli getpeerinfo
```

```
{
    "addr" : "85.213.199.39:8333",
    "services" : "00000001",
    "lastsend" : 1405634126,
    "lastrecv" : 1405634127,
    "bytessent" : 23487651,
    "bytesrecv" : 138679099,
    "conntime" : 1405021768,
    "pingtime" : 0.00000000,
    "version" : 70002,
    "subver" : "/Satoshi:0.9.2.1/",
    "inbound" : false,
    "startingheight" : 310131,
    "banscore" : 0,
    "syncnode" : true
},
{
    "addr" : "58.23.244.20:8333",
    "services" : "00000001",
    "lastsend" : 1405634127,
    "lastrecv" : 1405634124,
    "bytessent" : 4460918,
    "bytesrecv" : 8903575,
    "conntime" : 1405559628,
    "pingtime" : 0.00000000,
    "version" : 70001,
    "subver" : "/Satoshi:0.8.6/",
    "inbound" : false,
    "startingheight" : 311074,
    "banscore" : 0,
    "syncnode" : false
}
```

用户可以通过提供 -connect= 选项来指定一个或多个IP地址，从而达到覆盖自动节点管理功能并指定IP地址列表的目的。如果采用此选项，节点只连接到这些选定的节点IP地址，而不会自动发现并维护对等节点之间的连接。

如果已建立的连接没有数据通信，所在的节点会定期发送信息以维持连接。如果节点持续某个连接长达90分钟没有任何通信，它会被认为已经从网络中断开，网络将开始查找一个新的对等节点。因此，比特币网络会随时根据变化的节点及网络问题进行动态调整，不需经过中心化的控制即可进行规模增减的有机调整。

8.6 全节点

全节点是指维持包含全部交易信息的完整区块链的节点。更加准确地说，这样的节点应当被称为完整区块链节点”。在比特币发展的早期，所有节点都是全节点；当前的比特币核心客户端也是完整区块链节点。但在过去的两年中出现了许多 新型客户端，它们不需要维持完整的

区块链，而是作为轻量级客户端运行。在下面的章节里我们会对这些轻量级客户端进行详细介绍。

完整区块链节点保有完整的、最新的包含全部交易信息的比特币区块链拷贝，这样的节点可以独立地进行建立并校验区块链，从第一区块（创世区块）一直建立到网络中最新的区块。完整区块链节点可以独立自主地校验任何交易信息，而不需要借助任何其他节点或其他信息来源。完整区块节点通过比特币网络获取包含交易信息的新区块更新，在验证无误后将此更新合并至本地的区块链拷贝之中。

运行完整区块链节点可以给您一种纯粹的比特币体验：不需借助或信任其他系统即可独立地对所有交易信息进行验证。 辨别您是否在运行全节点是十分容易的：只需要查看您的永久性存储设备（如硬盘）是否有超过20GB的空间被用来存储完整区块链即可。如果您需要很大的磁盘空间、并且同步比特币网络耗时2至3天，那么您使用的正是全节点。这就是摆脱中心化管理、获得完全的独立自由所要付出的代价。

尽管目前还有一些使用不同编程语言及软件架构的其他的完整区块链客户端存在，但是最常用的仍然是比特币核心客户 端，它也被称为“Satoshi客户端”。比特币网络中超过90%的节点运行着各个版本的比特币核心客户端。如前文所述，它 可以通过节点间发送的version消息或通过getpeerinfo命令所得到的子版本字符串“Satoshi”加以辨识，例如 /Satoshi: 0.8.6/。

8.7 交换“库存清单”

一个全节点连接到对等节点之后，第一件要做的事情就是构建完整的区块链。如果该节点是一个全新节点，那么它就不包含任何区块链信息，它只知道一个区块——静态植入在客户端软件中的创世区块。新节点需要下载从0号区块（创世区块）开始的数十万区块的全部内容，才能跟网络同步、并重建全区块链。

同步区块链的过程从发送version消息开始，这是因为该消息中含有的BestHeight字段标示了一个节点当前的区块链高度（区块数量）。节点可以从它的对等节点中得到版本消息，了解双方各自有多少区块，从而可以与其自身区块链所拥 有的区块数量进行比较。对等节点们会交换一个getblocks消息，其中包含他们本地区块链的顶端区块哈希值（指纹）。如果某个对等节点识别出它接收到的哈希值并不属于顶端区块，而是属于一个非顶端区块的旧区块，那么它就能推断出：其自身的本地区块链比其他对等节点的区块链更长。

拥有更长区块链的对等节点比其他节点有更多的区块，可以识别出哪些区块们是其他节点需要“补充”的。它会识别出第 一批可供分享的500个区块，通过使用inv（inventory）消息把这些区块的哈希值传播出去。缺少这些区块的节点便可以 通过各自发送的getdata消息来请求得到全区块信息，用包含在inv消息中的哈希值来确认是否为正确的被请求的区块， 从而读取这些缺失的区块。

在下例中，我们假设某节点只含有创世区块。它收到了来自对等节点的inv消息，其中包含了区块链中后500个区块的哈希值。于是它开始向所有与之相连的对等节点请求区块，并通过分摊工作量的方式防止单一对等节点被批量请求所压垮。该节点会追踪记录其每个对等节点连接上“正在传输”（指那些它已经发出了请求但还没有接收到）的区块数量，并且检查该数量有没有超过上限（ MAX_BLOCKS_IN_TRANSIT_PER_PEER ）。用这种办法，如果一个节点需要更新大量区块，它会在上一请求完成后才发送对新区块的请求，从而允许对等节点控制更新速度，不至于压垮网络。每一个区块在被接收后就会被添加至区块链中，这一过程详见挖矿一章。随着本地区块链的逐步建立，越来越多的区块被请求和接收，整个过程将一直持续到该节点与全网络完成同步为止。

每当一个节点离线，不管离线时间有多长，这个与对等节点比较本地区块链并恢复缺失区块的过程就会被触发。如果一 个节点只离线几分钟，可能只会缺失几个区块；当它离线长达一个月，可能会缺失上千个区块。但无论哪种情况，它都 会从发送 getblocks 消息开始，收到一个inv响应，接着开始下载缺失的区块。库存清单和区块广播协议如图8-6所示。

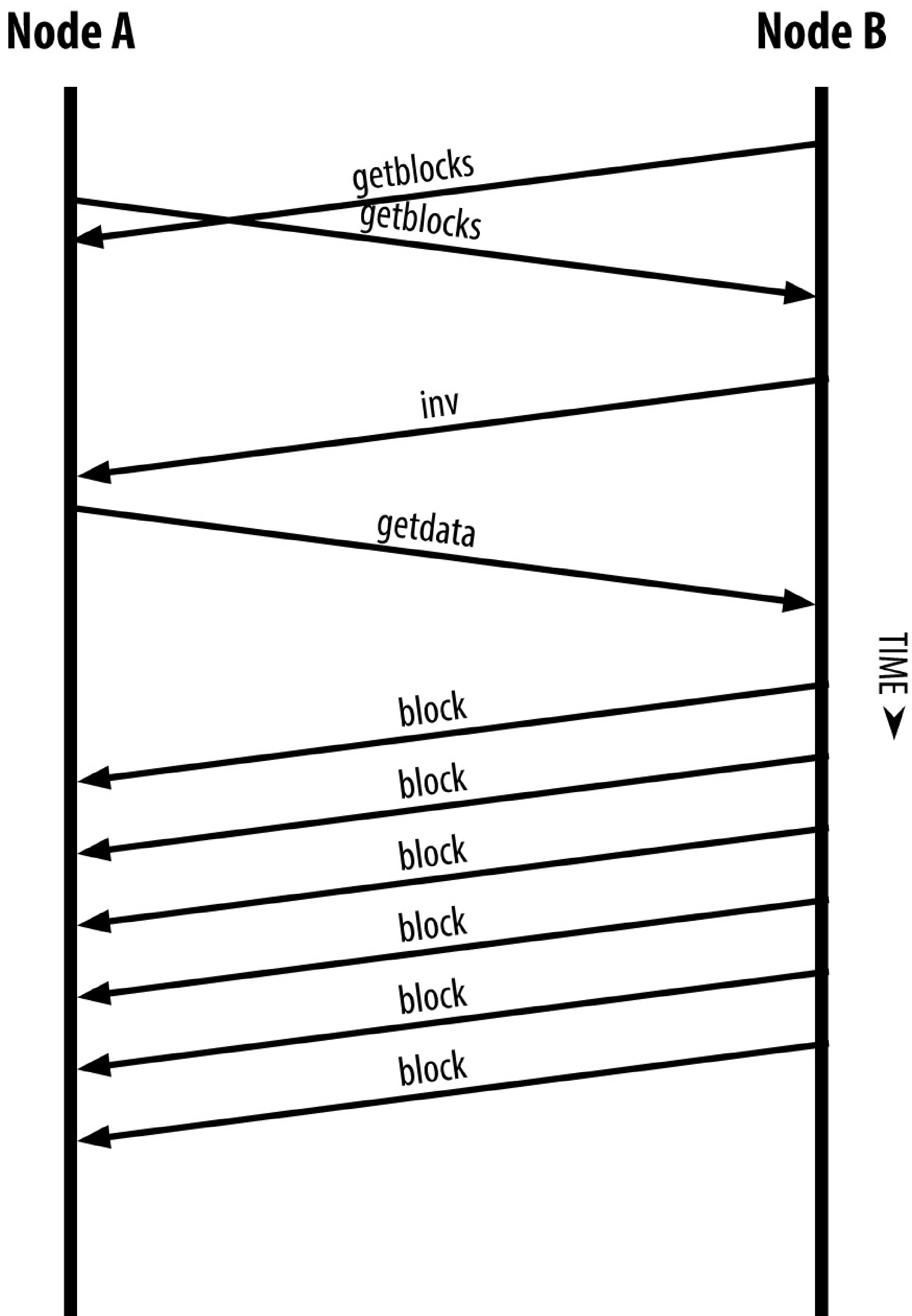


图8-6节点通过从对等体检索区块同步区块链

8.8 简易支付验证 （Simplified Payment Verification (SPV) ）节点

并非所有的节点都有能力储存完整的区块链。许多比特币客户端被设计成运行在空间和功率受限的设备上，如智能电话、平板电脑、嵌入式系统等。对于这样的设备，通过简化的支付验证（SPV）的方式可以使它们在不必存储完整区块链的情况下进行工作。这种类型的客端被称为SPV客户端或轻量级客户端。随着比特币的使用热潮，SPV节点逐渐变成比特币节点（尤其是比特币钱包）所采用的最常见的形式。

SPV节点只需下载区块头，而不用下载包含在每个区块中的交易信息。由此产生的不含交易信息的区块链，大小只有完整区块链的1/1000。SPV节点不能构建所有可用于消费的UTXO的全貌，这是由于它们并不知道网络上所有交易的完整信息。SPV节点验证交易时所使用的方法略有不同，这个方法需依赖对等节点“按需”提供区块链相关部分的局部视图。

打个比方来说，每个全节点就像是一个在陌生城市里的游客，他带着一张包含每条街道、每个地址的详细地图。相比之 下，SPV节点就像是这名陌生城市里的游客只知道一条主干道的名字，通过随机询问该城市的陌生人来获取分段道路指示。虽然两种游客都可以通过实地考察来验证一条街是否存在，但没有地图的游客不知道每个小巷中有哪些街道，也不知道附近还有什么其他街道。没有地图的游客在“教堂街23号”的前面，并不知道这个城市里是否还有其他若干条“教堂街 23号”，也不知道面前的这个是否是要找的那个。对他来说，最好的方式就是向足够多的人问路，并且希望其中一部分人不是要试图抢劫他。

简易支付验证是通过参考交易在区块链中的深度，而不是高度，来验证它们。一个拥有完整区块链的节点会构造一条验证链，这条链是由沿着区块链按时间倒序一直追溯到创世区块的数千区块及交易组成。而一个SPV节点会验证所有区块的链（但不是所有的交易），并且把区块链和有关交易链接起来。

例如，一个全节点要检查第300,000号区块中的某个交易，它会把从该区块开始一直回溯到创世区块的300,000个区块全部都链接起来，并建立一个完整的UTXO数据库，通过确认该UTXO是否还未被支付来证实交易的有效性。SPV节点则不能验证UTXO是否还未被支付。相反地，SPV节点会在该交易信息和它所在区块之间用merkle路径（见“ Merkle 树”章节）建立一条链接。然后SPV节点一直等待，直到序号从300,001到300,006的六个区块堆叠在该交易所在的区块之上，并通过确立交易的深度是在第300,006区块~第300,001区块之下来验证交易的有效性。事实上，如果网络中的其他节点都接受了第300,000区块，并通过足够的工作在该块之上又生成了六个区块，根据代理网关协议，就可以证明该交易不是双重支付。

如果一个交易实际上不存在，SPV节点不会误认为该交易存在于某区块中。SPV节点会通过请求merkle路径证明以及验证区块链中的工作量证明，来证实交易的存在性。可是，一个交易的存在是可能对SPV节点“隐藏”的。SPV节点毫无疑问可以证实某个交易的存在性，但它不能验证某个交易（譬如同一个UTXO的双重支付）不存在，这是因为SPV节点没有一份关于所有交易的记录。这个漏洞会被针对SPV节点的拒绝服务攻击或双重支付型攻击所利用。为了防御这些攻击，SPV节点需要随机连接到多个节点，以增加与至少一个可靠节点相连接的概

率。这种随机连接的需求意味着SPV节 点也容易受到网络分区攻击或Sybil攻击。在后者情况中，SPV节点被连接到虚假节点或虚假网络中，没有通向可靠节点或真正的比特币网络的连接。

在绝大多数的实际情况中，具有良好连接的SPV节点是足够安全的，它在资源需求、实用性和安全性之间维持恰当的平衡。当然，如果要保证万无一失的安全性，最可靠的方法还是运行完整区块链的节点。

提示 完整的区块链节点是通过检查整个链中在它之下的数千个区块来保证这个UTXO没有被支付，从而验证交易。而 SPV节点是通过检查在其上面的区块将它压在下面的深度来验证交易。

SPV节点使用的是一条getheaders消息，而不是getblocks消息来获得区块头。发出响应的对等节点将用一条headers 消息发送多达2000个区块头。这一过程和全节点获取所有区块的过程没什么区别。SPV节点还在与对等节点的连接上设置了过滤器，用以过滤从对等节点发来的未来区块和交易数据流。任何目标交易都是通过一条getdata的请求来读取的。对等节点生成一条包含交易信息的tx消息作为响应。区块头的同步过程如图8-7所示。

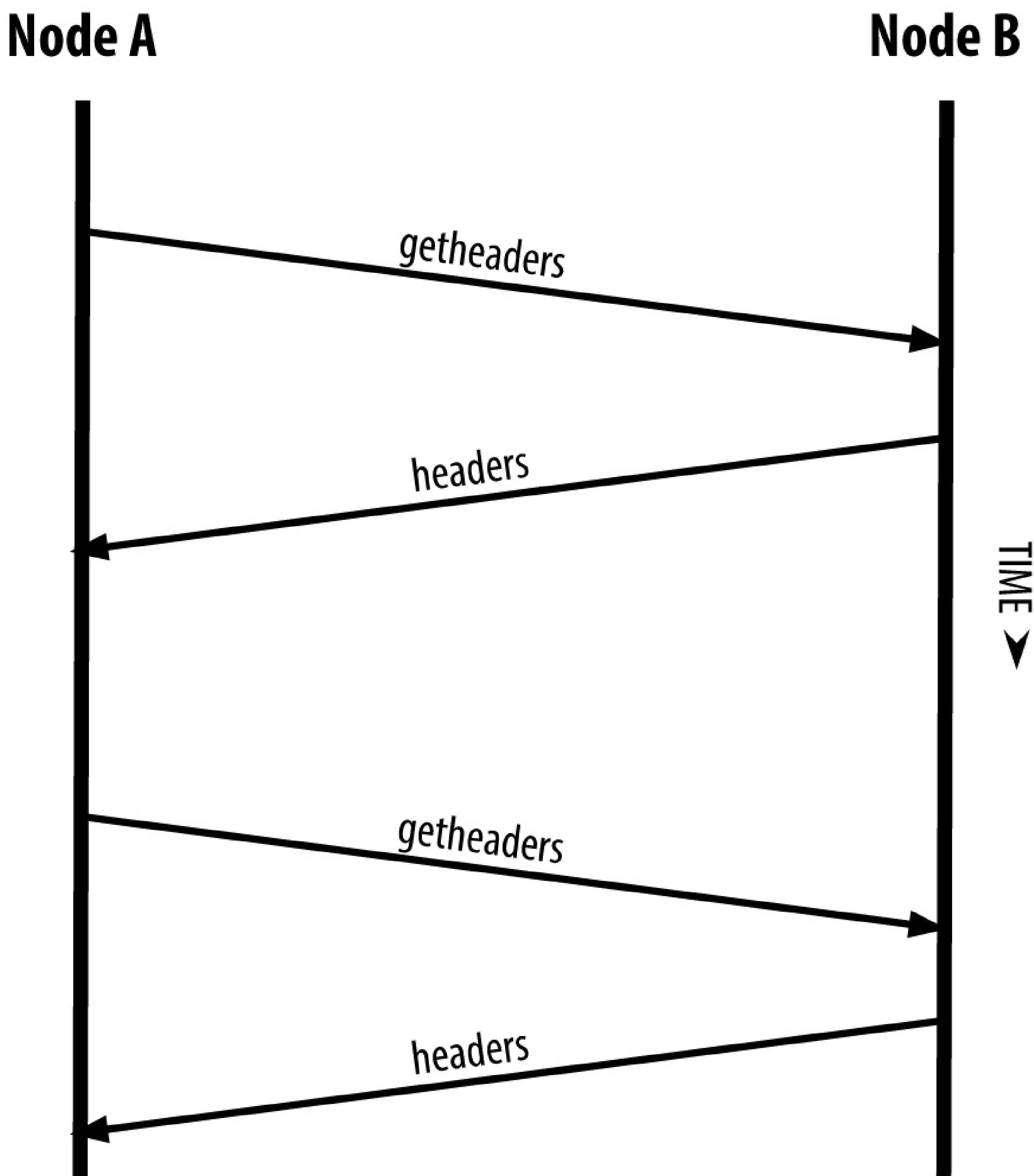


图8-7SPV节点同步区块头

由于SPV节点需要读取特定交易从而选择性地验证交易，这样就又产生了隐私风险。与全区块链节点收集每一个区块内的全部交易所不同的是，SPV节点对特定数据的请求可能无意中透露了钱包里的地址信息。例如，监控网络的第三方可以跟踪某个SPV节点上的钱包所请求的全部交易信息，并且利用这些交易信息把比特币地址和钱包的用户关联起来，从而损害了用户的隐私。

在引入SPV节点/轻量级节点后不久，比特币开发人员就添加了一个新功能：Bloom过滤器，用以解决SPV节点的隐私风 险问题。Bloom过滤器通过一个采用概率而不是固定模式的过滤机制，允许SPV节点只接收交易信息的子集，同时不会精确泄露哪些是它们感兴趣的地址。

在引入SPV /轻量级节点后不久，比特币开发人员添加了一个名为bloom过滤器的功能来解决SPV节点的隐私风险。 Bloom过滤器允许SPV节点接收交易的一个子集，通过使用概率而不是固定模式的过滤机制无需精确地揭示他们感兴趣的地址。

8.9 Bloom过滤器

Bloom过滤器是一个允许用户描述特定的关键词组合而不必精确表述的基于概率的过滤方法。它能让用户在有效搜索关键词的同时保护他们的隐私。在SPV节点里，这一方法被用来向对等节点发送交易信息查询请求，同时交易地址不会被 暴露。

用我们之前的例子，一位手中没有地图的游客需要询问去特定地方的路线。如果他向陌生人询问“教堂街23号在哪里”， 不经意之间，他就暴露了自己的目的地。Bloom过滤器则会这样问，附近有带‘堂’字的街道吗？”这样的问法包含了比之前略少的关键词。这位游客可以自己选择包含信息的多少，比如“以‘堂街’结尾”或者“‘教’字开头的街道”。如果他问得越少，得到了更多可能的地址，隐私得到了保护，但这些地址里面不乏无关的结果；如果他问得非常具体，他在得到较准 确的结果的同时也暴露了自己的隐私。

Bloom过滤器可以让SPV节点指定交易的搜索模式，该搜索模式可以基于准确性或私密性的考虑被调节。一个非常具体 的Bloom过滤器会生成更准确的结果，但也会显示该用户钱包里的使用的地址；反之，如果过滤器只包含简单的关键 词，更多相应的交易会被搜索出来，在包含若干无关交易的同时有着更高的私密性。

8.9.1 Bloom过滤器如何工作

Bloom过滤器的实现是由一个可变长度（N）的二进制数组（N位二进制数构成一个位域）和数量可变（M）的一组哈希函数组成。这些哈希函数的输出值始终在1和N之间，该数值与二进制数组相对应。并且该函数为确定性函数，也就是说任何一个使用相同Bloom过滤器的节点通过该函数都能对特定输入得到同一个的结果。Bloom过滤器的准确性和私密 性能通过改变长度（N）和哈希函数的数量（M）来调节。

在图8-8中，我们用一个小型的十六位数组和三个哈希函数来演示Bloom过滤器的应用原理。

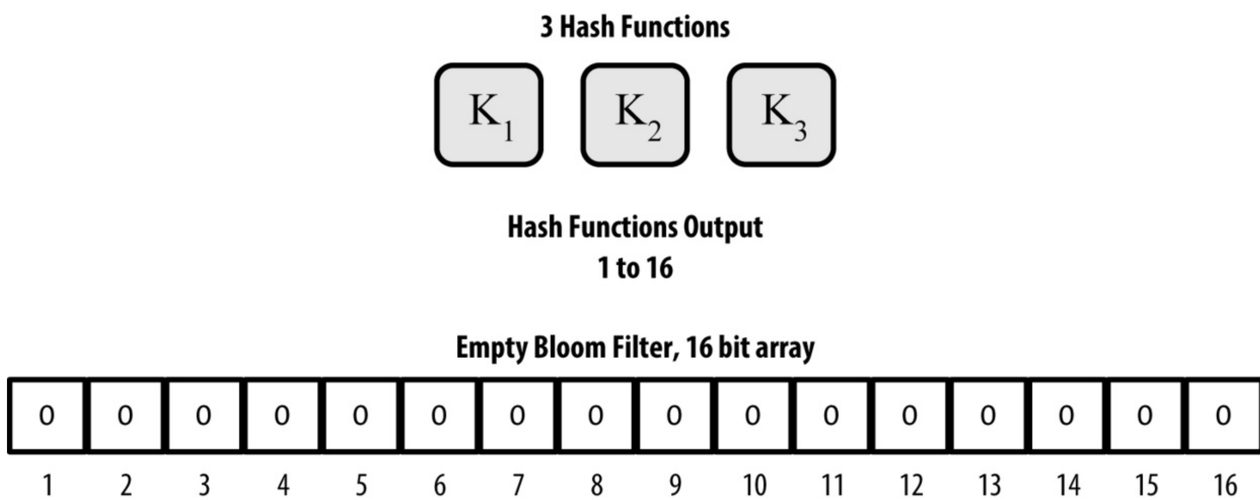


图8-8一个简单的Bloom过滤器的例子，有一个16位的字段和三个哈希函数

Bloom过滤器数组里的每一个数的初始值为零。关键词被加到Bloom过滤器中之前，会依次通过每一个哈希函数运算一 次。该输入经第一个哈希函数运算后得到了一个在1和N之间的数，它在该数组（编号依次为1至N）中所对应的位被置 为1，从而把哈希函数的输出记录下来。接着再进行下一个哈希函数的运算，把另外一位置为1；以此类推。当全部M个 哈希函数都运算过之后，一共有M个位的值从0变成了1，这个关键词也被“记录”在了Bloom过滤器里。

图8-9显示了向图8-8里的简易Bloom过滤器添加关键词“A”。

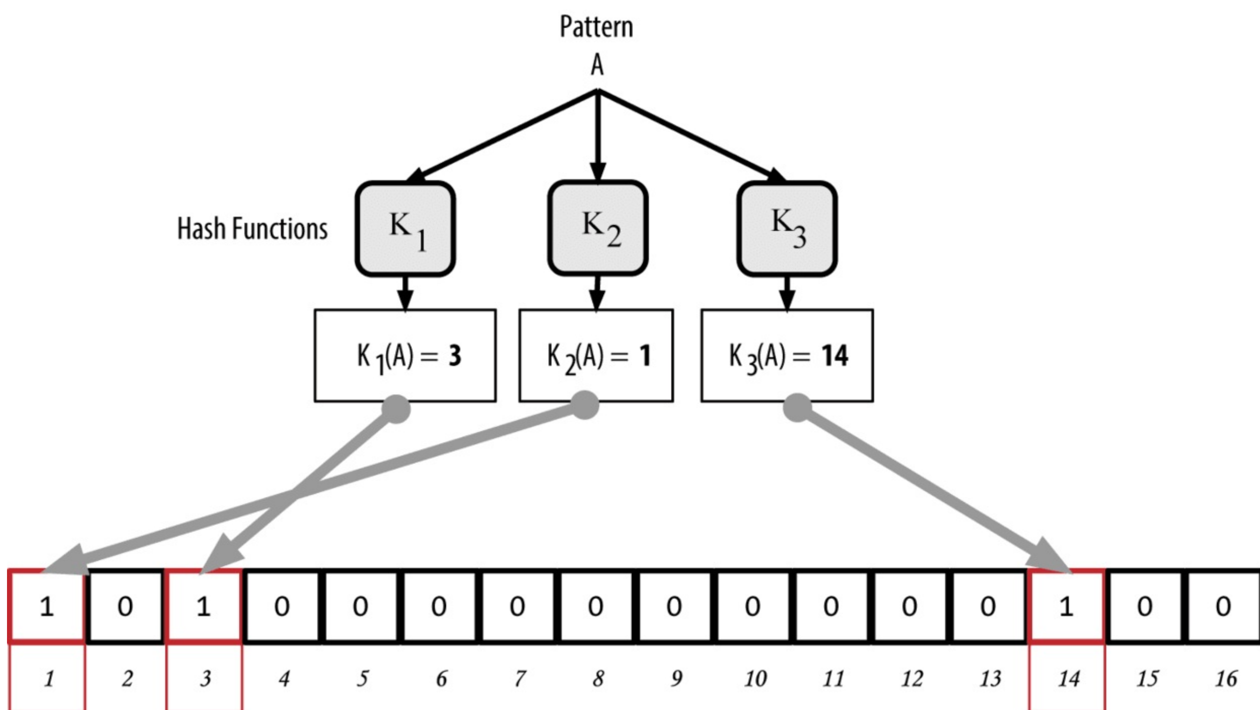


图8-9向简易Bloom过滤器添加关键词“A”

增加第二个关键是就是简单地重复之前的步骤。关键词依次通过各哈希函数运算之后，相应的位变为1，Bloom过滤器 则记录下该关键词。需要注意的是，当Bloom过滤器里的关键词增加时，它对应的某个哈希函数的输出值的位可能已经 是1了，这种情况下，该位不会再次改变。也就是说，随着更多的关键词指向了重复的位，Bloom过滤器随着位1的增加 而饱和，准

确性也因此降低了。该过滤器之所以是基于概率的数据结构，就是因为关键词的增加会导致准确性的降低。 准确性取决于关键字的数量以及数组大小（N）和哈希函数的多少（M）。更大的数组和更多的哈希函数会记录更多的 关键词以提高准确性。而小的数组及有限的哈希函数只能记录有限的关键词从而降低准确性。

图8-10显示了向该简易Bloom过滤器里增加第二个关键词“B”。

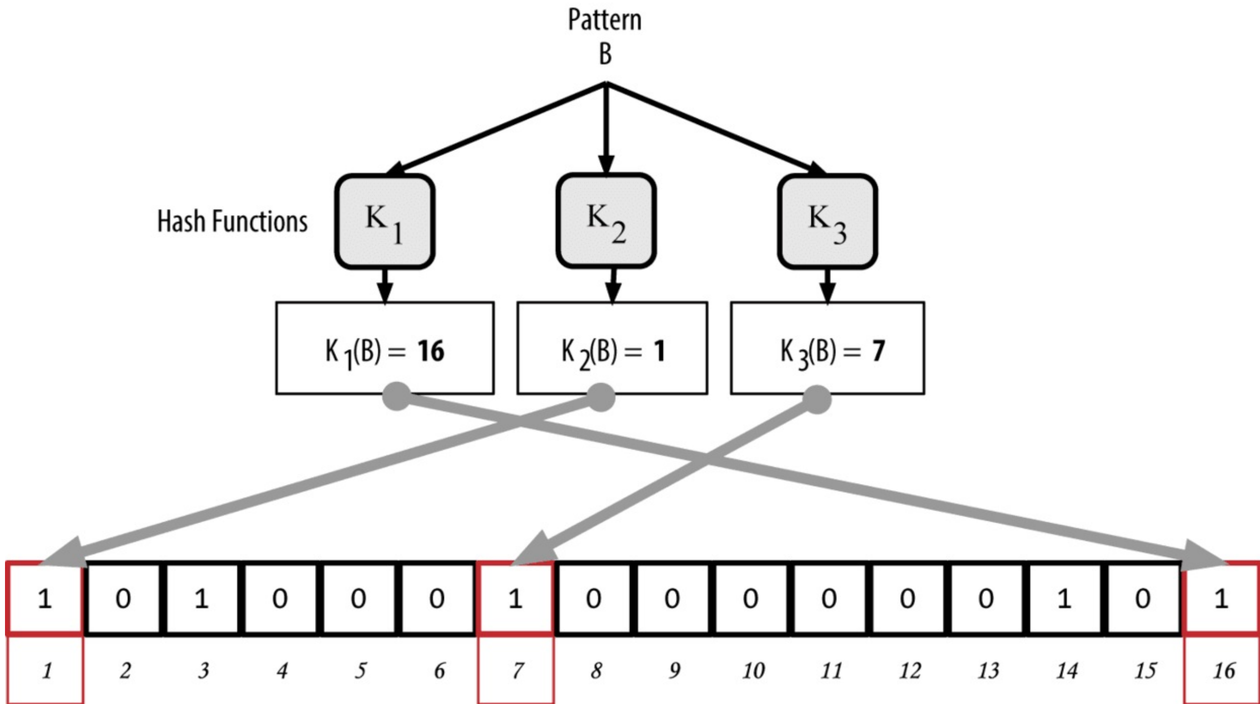


图8-10向该简易Bloom过滤器里增加第二个关键词“B”

为测试某一关键词是否被记录在某个Bloom过滤器中，我们将该关键词逐一代入各哈希函数中运算，并将所得的结果与原数组进行对比。如果所有的结果对应的位都变为了1，则表示这个关键词有可能已被该过滤器记录。之所以这一结论并不确定，是因为这些字节1也有可能是其他关键词运算的重叠结果。简单来说，Bloom过滤器正匹配代表着“可能是”。 图8-11是一个验证关键词“X”是否在前述Bloom过滤器中的图例。相应的比特位都被置为1，所以这个关键词很有可能是匹配的。

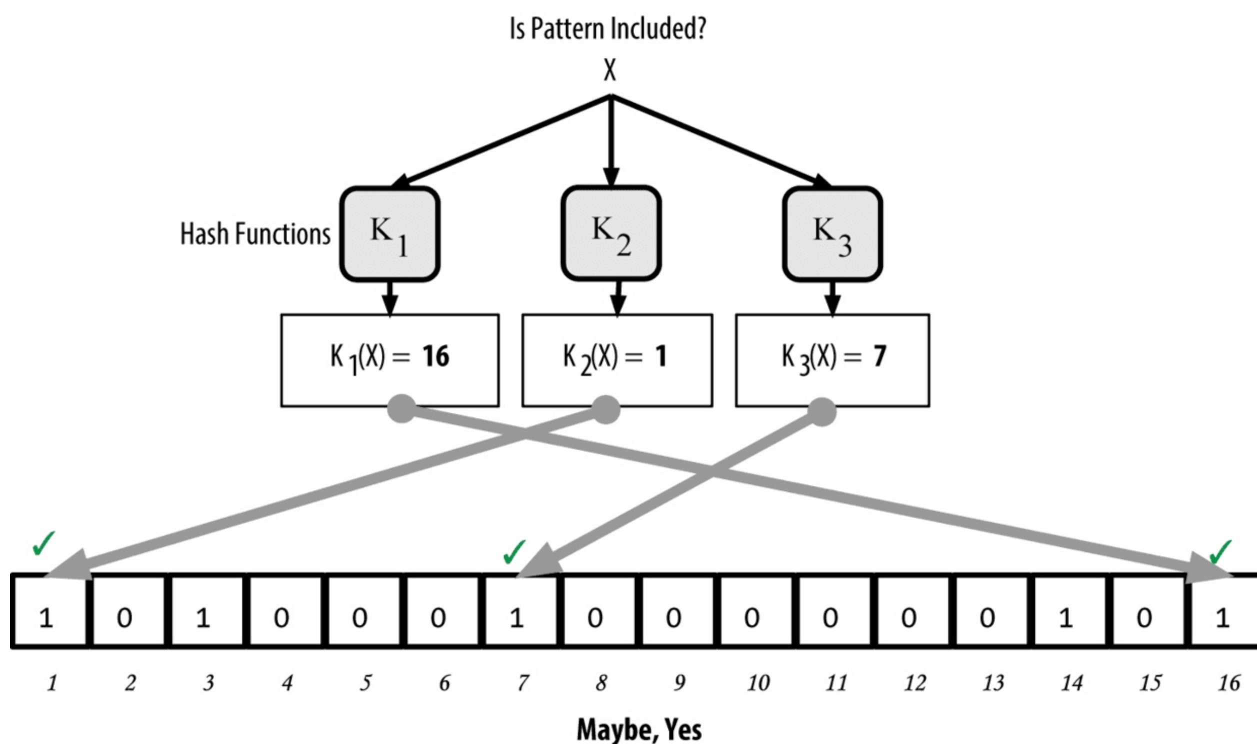


图8-11在bloom过滤器中测试模式“X”的存在。 结果是一个概率正匹配，意思是“也许”

另一方面，如果我们代入关键词计算后的结果某位为0，说明该关键词并没有被记录在过滤器里。负匹配的结果不是可 能，而是一定。也就是说，负匹配代表着“一定不是”。 图8-12是一个验证关键词“Y”是否存在于简易Bloom过滤器中的图例。图中某个结果字段为0，该字段一定没有被匹配。

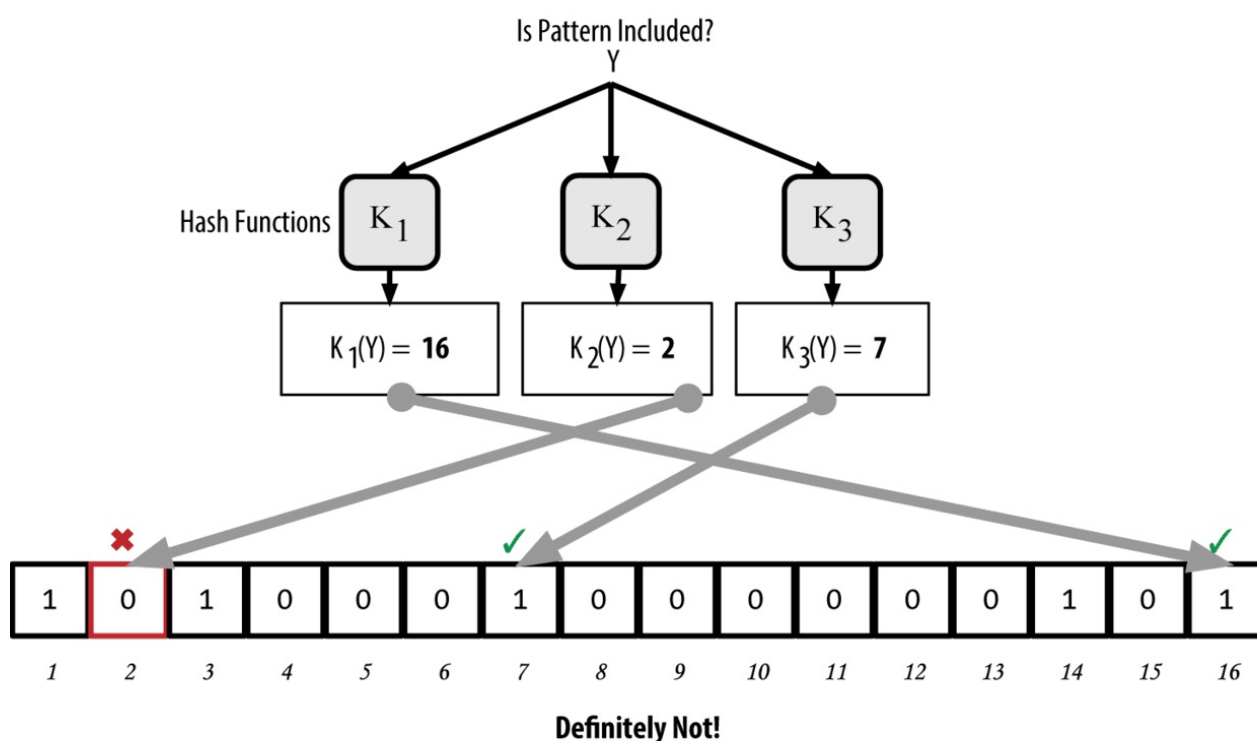


图8-12在bloom过滤器中测试模式“Y”的存在。 结果是一个明确的否定匹配，意思是“绝对不！”

8.10 SPV节点如何使用Bloom过滤器

Bloom过滤器用于过滤SPV节点从其对等体接收的交易（和包含它们的块），仅选择SPV节点感兴趣的交易，而不会泄露其感兴趣的地址或密钥。

SPV节点将初始化“过滤器”为“空”;在该状态下，bloom过滤器将不匹配任何模式。然后，SPV节点将列出所有感兴趣的地址，密钥和散列，它将通过从其钱包控制的任何UTXO中提取公钥哈希和脚本哈希和交易ID来实现。 SPV节点然后将其中的每一个添加到Bloom过滤器，以便如果这些模式存在于交易中，则Bloom过滤器将“匹配”，而不会自动显示模式。

然后，SPV节点将向对等体发送一个过滤器加载消息，其中包含在连接上使用的bloom过滤器。在对等体上，针对每个传入交易检查Bloom过滤器。完整节点根据bloom过滤器检查交易的几个部分，寻找匹配，包括：

交易ID

每个交易输出的锁定脚本的数据组件（脚本中的每个键和哈希）

每个交易输入

每个输入签名数据组件（或见证脚本）

通过检查所有这些组件，可以使用Bloom过滤器来匹配公钥哈希，脚本，OP_RETURN值，签名中的公钥或智能合同或复杂脚本的任何未来组件。

在建立过滤器之后，对等体然后将针对bloom过滤器测试每个交易的输出。只有与过滤器匹配的交易才会发送到节点。

响应于来自节点的getdata消息，对等体将发送一个merkleblock消息，该消息仅包含与过滤器匹配的块和每个匹配交易的merkle路径（参见[merkle_trees]）的块头。然后，对等体还将发送包含由过滤器匹配的交易的tx消息。

由于完整节点向SPV节点发送交易，SPV节点丢弃任何误报，并使用正确匹配的交易来更新其UTXO集和钱包余额。随着它更新自己的UTXO集视图，它还会修改bloom过滤器，以匹配任何引用其刚刚发现的UTXO的交易。然后，完整节点使用新的bloom过滤器来匹配新交易，并重复整个过程。

设置bloom过滤器的节点可以通过发送filteradd消息将模式交互式添加到过滤器。要清除bloom过滤器，节点可以发送一个过滤清除消息。因为不可能从布局过滤器中删除模式，所以如果不再需要模式，则节点必须清除并重新发送新的布隆过滤器。

BIP-37 (Peer Services)中定义了SPV节点的网络协议和布隆过滤机制。

8.11 SPV节点和隐私

实现SPV的节点的隐私比整个节点更弱。完整节点接收所有交易，因此不会显示关于它的钱包中是否使用某个地址的信息。 SPV节点接收与其钱包中的地址相关的经过过滤的列表。结果，它减少了所有者的隐私。

bloom过滤器是减少隐私损失的一种方式。没有它们，SPV节点将不得不明确地列出它感兴趣的地址，造成严重的隐私违规。然而，即使使用过滤器，对手监控SPV客户端的流量或直接连接到它的P2P网络中的节点可以随时随地收集足够的信息来了解SPV客户端的钱包中的地址。

8.12加密和认证连接

比特币的大多数新用户假设比特币节点的网络通信是加密的。其实，比特币的原始实现就很明显地完成了。虽然这不是完整节点的主要隐私问题，但SPV节点是一个很大的问题。

作为增加比特币P2P网络隐私和安全性的一种方法，有两种解决方案可以通过BIP-150/151提供通信加密：Tor传输和P2P认证和加密。

8.12.1Tor网络传输

Tor代表洋葱路由网络，是一个软件项目和网络，通过提供匿名，不可追踪和隐私的随机网络路径提供数据的加密和封装。

Bitcoin Core提供了多种配置选项，允许您运行通过Tor网络传输的流量的比特币节点。此外，Bitcoin Core还可以提供Tor隐藏服务，允许其他Tor节点通过Tor直接连接到您的节点。

从Bitcoin Core版本0.12开始，如果能够连接到本地Tor服务，节点将自动提供隐藏的Tor服务。如果您安装Tor并且Bitcoin Core进程作为具有足够权限的用户访问Tor认证cookie的用户运行，则应自动运行。使用debug标志打开Bitcoin Core对于Tor服务的调试，如下所示：

$ bitcoind --daemon --debug=tor

你应该在日志中看到“tor：ADD_ONION success”，表示Bitcoin Core已经向Tor网络添加了隐藏的服务。

您可以在Bitcoin Core文档（docs / tor.md）和各种在线教程中找到有关运行Bitcoin Core作为Tor隐藏服务的更多说明。

8.12.2对等认证和加密

BIP-150和BIP-151两种比特币改进方案在比特币P2P网络中增加了对P2P认证和加密的支持。这两个BIP定义了可由兼容的比特币节点提供的可选服务。 BIP-151启用了支持BIP-151的两个节点之间的所有通信的协商加密。 BIP-150提供可选的对等认证，允许节点使用

ECDSA和私钥对对方的身份进行身份验证。 BIP-150要求在认证之前，两个节点按照BIP-151建立了加密通信。

截至2017年1月，BIP-150和BIP-151未在Bitcoin Core中实施。但是，这两个提案已由至少一个名为bcoin的替代比特币客户端实施。

BIP-150和BIP-151允许用户运行连接到受信任的完整节点的SPV客户端，使用加密和身份验证来保护SPV客户端的隐私。

此外，可以使用身份验证来创建可信比特币节点的网络，并防止中间人攻击。最后，P2P加密（如果广泛部署）将加强比特币对流量分析和隐私侵权监控的阻力，特别是在互联网使用受到严格控制和监控的极权主义国家。

标准定义在BIP-150 (Peer Authentication) and BIP-151 (Peer-to-Peer Communication Encryption).

8.13交易池

比特币网络中几乎每个节点都会维护一份未确认交易的临时列表，被称为内存池或交易池。节点们利用这个池来追踪记录那些被网络所知晓、但还未被区块链所包含的交易。例如，保存用户钱包的节点会利用这个交易池来记录那些网络已经接收但还未被确认的、属于该用户钱包的预支付信息。

随着交易被接收和验证，它们被添加到交易池并通知到相邻节点处，从而传播到网络中。

有些节点的实现还维护一个单独的孤立交易池。如果一个交易的输入与某未知的交易有关，如与缺失的父交易相关，该 孤立交易就会被暂时储存在孤立交易池中直到父交易的信息到达。

当一个交易被添加到交易池中，会同时检查孤立交易池，看是否有某个孤立交易引用了此交易的输出（子交易）。任何 匹配的孤立交易会被进行验证。如果验证有效，它们会从孤立交易池中删除，并添加到交易池中，使以其父交易开始的链变得完整。对新加入交易池的交易来说，它不再是孤立交易。前述过程重复递归寻找进一步的后代，直至所有的后代都被找到。通过这一过程，一个父交易的到达把整条链中的孤立交易和它们的父级交易重新结合在一起，从而触发了整 条独立交易链进行级联重构。

交易池和孤立交易池（如有实施）都是存储在本地内存中，并不是存储在永久性存储设备（如硬盘）里。更准确的说， 它们是随网络传入的消息动态填充的。节点启动时，两个池都是空闲的；随着网络中新交易不断被接收，两个池逐渐被 填充。

有些比特币客户端的实现还维护一个UTXO数据库，也称UTXO池，是区块链中所有未支付交易输出的集合。“UTXO 池”的名字听上去与交易池相似，但它代表了不同的数据集。UTXO池不同于交易池和孤立交易池的地方在于，它在初始 化时不为空，而是包含了数以百万计的未

支付交易输出条目，有些条目的历史甚至可以追溯至2009年。UTXO池可能会被安置在本地内存，或者作为一个包含索引的数据库表安置在永久性存储设备中。

交易池和孤立交易池代表的是单个节点的本地视角。取决于节点的启动时间或重启时间，不同节点的两池内容可能有很 大差别。相反地，UTXO池代表的是网络的突显共识，因此，不同节点间UTXO池的内容差别不大。此外，交易池和孤立交易池只包含未确认交易，而UTXO池之只包含已确认交易。

9.1简介

区块链的数据结构是由包含交易信息的区块按照从远及近的顺序有序链接起来的。它可以被存储为平面文件（flat file），或是存储在一个简单数据库中。比特币核心客户端使用Google的LevelDB数据库存储区块链元数据。区块被从远及近有序地链接在这个链条里，每个区块都指向前一个区块。区块链经常被视为一个垂直的栈，第一个区块作为栈底的首区块，随后每个区块都被放置在之前的区块之上。用栈来形象化表示区块依次堆叠这一概念后，我们便可以使用 一些术语，例如：“高度”来表示区块与首区块之间的距离；以及“顶部”或“顶端”来表示最新添加的区块。

对每个区块头进行SHA256加密哈希，可生成一个哈希值。通过这个哈希值，可以识别出区块链中的对应区块。同时， 每一个区块都可以通过其区块头的“父区块哈希值”字段引用前一区块（父区块）。也就是说，每个区块头都包含它的父区块哈希值。这样把每个区块链接到各自父区块的哈希值序列就创建了一条一直可以追溯到第一个区块（创世区块）的 链条。

虽然每个区块只有一个父区块，但可以暂时拥有多个子区块。每个子区块都将同一区块作为其父区块，并且在“父区块哈希值”字段中具有相同的（父区块）哈希值。一个区块出现多个子区块的情况被称为“区块链分叉”。区块链分叉只是暂时状态，只有当多个不同区块几乎同时被不同的矿工发现时才会发生（参见“区块链分叉”）。最终，只有一个子区块会成为区块链的一部分，同时解决了“区块链分叉”的问题。尽管一个区块可能会有不止一个子区块，但每一区块只有 一个父区块，这是因为一个区块只有一个“父区块哈希值”字段可以指向它的唯一父区块。

由于区块头里面包含“父区块哈希值”字段，所以当前区块的哈希值也受到该字段的影响。如果父区块的身份标识发生变化，子区块的身份标识也会跟着变化。当父区块有任何改动时，父区块的哈希值也发生变化。这将迫使子区块的“父区块哈希值”字段发生改变，从而又将导致子区块的哈希值发生改变。而子区块的哈希值发生改变又将迫使孙区块的“父区块哈希值”字段发生改变，又因此改变了孙区块哈希值，以此类推。一旦一个区块有很多代以后，这种瀑布效应将保证该区块不会被改变，除非强制重新计算该区块所有后续的区块。正是这样的重新计算需要耗费巨大的计算量，所以一个长区块链的存在可以让区块链的历史不可改变，这也是比特币安全性的一个关键特征。

你可以把区块链想象成地质构造中的地质层或者是冰川岩芯样品。表层可能会随着季节而变化，甚至在沉积之前就被风吹走了。但是越往深处，地质层就变得越稳定。到了几百英尺深的地方，你看到的将是保存了数百万年但依然保持历史原状的岩层。在区块链里，最近的几个区块可能会由于区块链分叉所引发的重新计算而被修改。最新的六个区块就像几英寸深的表土层。但是，超过这六块后，区块在区块链中的位置越深，被改变的可能性就越小。在100个区块以后，区块链已经足够稳定，这时Coinbase交易（包含新挖出的比特币的交易）可以被支付。几千个区块（一个月）后的区块链将变成确定的历史，永远不会改变。

9.2 区块结构

区块是一种被包含在公开账簿（区块链）里的聚合了交易信息的容器数据结构。它由一个包含元数据的区块头和紧跟其后的构成区块主体的一长串交易列表组成。区块头是80字节，而平均每个交易至少是250字节，而且平均每个区块至少包含超过500个交易。因此，一个包含所有交易的完整区块比区块头大1000倍。表7-1描述了一个区块结构。

Size	Field	Description
4 bytes	Block Size	The size of the block, in bytes, following this field
80 bytes	Block Header	Several fields form the block header
1–9 bytes (VarInt)	Transaction Counter	How many transactions follow
Variable	Transactions	The transactions recorded in this block

9.3 区块头

区块头由三组区块元数据组成。首先是一组引用父区块哈希值的数据，这组元数据用于将该区块与区块链中前一区块相连接。第二组元数据，即难度、时间戳和nonce，与挖矿竞争相关，详见挖矿章节。第三组元数据是merkle树根（一种用来有效地总结区块中所有交易的数据结构）。表7-2描述了区块头的数据结构。

Size	Field	Description
4 bytes	Version	A version number to track software/protocol upgrades
32 bytes	Previous Block Hash	A reference to the hash of the previous (parent) block in the chain
32 bytes	Merkle Root	A hash of the root of the merkle tree of this block’s transactions
4 bytes	Timestamp	The approximate creation time of this block (seconds from Unix Epoch)
4 bytes	Difficulty Target	The Proof-of-Work algorithm difficulty target for this block
4 bytes	Nonce	A counter used for the Proof-of-Work algorithm

Nonce、难度目标和时间戳会用于挖矿过程，更多细节将在挖矿章节讨论。

9.4 区块标识符：区块头哈希值和区块高度

区块主标识符是它的加密哈希值，一个通过SHA256算法对区块头进行二次哈希计算而得到的数字指纹。产生的32字节哈希值被称为区块哈希值，但是更准确的名称是：区块头哈希值，因为只有区块头被用于计算。例如:000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f是第一个比特币区块的区块哈希值。区块哈希值可以唯一、明确地标识一个区块，并且任何节点通过简单地对区块头进行哈希计算都可以独立地获取该区块哈希值。

请注意，区块哈希值实际上并不包含在区块的数据结构里，不管是该区块在网络上传输时，抑或是它作为区块链的一部分被存储在某节点的永久性存储设备上时。相反，区块哈希值是当该区块从网络被接收时由每个节点计算出来的。区块的哈希值可能会作为区块元数据的一部分被存储在一个独立的数据库表中，以便于索引和更快地从磁盘检索区块。

第二种识别区块的方式是通过该区块在区块链中的位置，即“区块高度（block height）”。第一个区块，其区块高度为 0，和之前哈希值000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f所引用的区块为同一 个区块。因此，区块可以通过两种方式被识别：区块哈希值或者区块高度。每一个随后被存储在第一个区块之上的区块在区块链中都比前一区块“高”出一个位置，就像箱子一样，一个接一个堆叠在其他箱子之上。2017年1月1日的区块高度大约是 446,000，说明已经有446,000个区块被堆叠在2009年1月创建的第一个区块之上。

和区块哈希值不同的是，区块高度并不是唯一的标识符。虽然一个单一的区块总是会有一个明确的、固定的区块高度， 但反过来却并不成立，一个区块高度并不总是识别一个单一的区块。两个或两个以上的区块可能有相同的区块高度，在区块链里争夺同一位置。这种情况在“区块链分叉”一节中有详细讨论。区块高度也不是区块数据结构的一部分， 它并不被存储在区块里。当节点接收来自比特币网络的区块时，会动态地识别该区块在区块链里的位置（区块高度）。 区块高度也可作为元数据存储在一个索引数据库表中以便快速检索。

提示一个区块的区块哈希值总是能唯一地识别出一个特定区块。一个区块也总是有特定的区块高度。但是，一 个特定的区块高度并不一定总是能唯一地识别出一个特定区块。更确切地说，两个或者更多数量的区块也许会为了区块链中的一个位置而竞争。

9.5 创世区块

区块链里的第一个区块创建于2009年，被称为创世区块。它是区块链里面所有区块的共同祖先，这意味着你从任一区块，循链向后回溯，最终都将到达创世区块。 因为创世区块被编入到比特币客户端软件里，所以每一个节点都始于至少包含一个区块的区块链，这能确保创世区块不会被改变。每一个节点都“知道”创世区块的哈希值、结构、被创建的时间和里面的一个交易。因此，每个节点都把该区块作为区块链的首区块，从而构建了一个安全的、可信的区块链。 在*chainparams.cpp*里可以看到创世区块被编入到比特币核心客户端里。

创世区块的哈希值为：

0000000000 19d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

你可以在任何区块浏览网站搜索这个区块哈希值，如blockchain.info，你会发现一个描述这一区块内容的页面，该页面的链接包含了这个区块哈希值：

https://blockchain.info/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
https://blockexplorer.com/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

在命令行使用比特币核心客户端：

``$ bitcoin-cli getblock
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

{ "hash" : "000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
"confirmations" : 308321,
"size" : 285,
"height" : 0,
"version" : 1,
"merkleroot" :
"4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b",
"tx" : ["4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b"],
"time" : 1231006505,
"nonce" : 2083236893,
"bits" : "1d00ffff",
"difficulty" : 1.00000000,
"nextblockhash" :
"00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048"}``

创世区块包含一个隐藏的信息。在其Coinbase交易的输入中包含这样一句话"The Times 03/Jan/2009 Chancellor on brink of second bailout forbanks."这句话是泰晤士报当天的头版文章标题，引用这句话，既是对该区块产生时间的说明，也可视为半开玩笑地提醒人们一个独立的货币制度的重要性，同时告诉人们随着比特币的发展，一场前所未有的世界性货币革命将要发生。该消息是由比特币的创立者中本聪嵌入创世区块中。

9.6 区块链接成为区块链

比特币的全节点在本地保存了区块链从创世区块起的完整副本。随着新的区块的产生，该区块链的本地副本会不断地更新用于扩展这个链条。当一个节点从网络接收传入的区块时，它会验证这些区块，然后链接到现有的区块链上。为建立一个连接，一个节点将检查传入的区块头并寻找该区块的"父区块哈希值"。

让我们假设，例如，一个节点在区块链的本地副本中有277,314个区块。该节点知道最后一个区块为第277,314个区 块，这个区块的区块头哈希值为：

00000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249。

然后该比特币节点从网络上接收到一个新的区块，该区块描述如下：

`{

```
"size" : 43560,
"version" : 2,
"previousblockhash" :
    "00000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249",
"merkleroot" :
    "5e049f4030e0ab2debb92378f53c0a6e09548aea083f3ab25e1d94ea1155e29d",
"time" : 1388185038,
"difficulty" : 1180923195.25802612,
"nonce" : 4215469401,
"tx" : [
    "257e7497fb8bc68421eb2c7b699dbab234831600e7352f0d9e6522c7cf3f6c77",
```

[... many more transactions omitted ...]

```
    "05cfd38f6ae6aa83674cc99e4d75a1458c165b7ab84725eda41d018a09176634"
]
```

}`

对于这一新的区块，节点会在“父区块哈希值”字段里找出包含它的父区块的哈希值。这是节点已知的哈希值，也就是第 277314块区块的哈希值。故这个新区块是这个链条里的最后一个区块的子区块，因此现有的区块链得以扩展。节点将新的区块添加至链条的尾端，使区块链变长到一个新的高度277,315。图9-1显示了通过“父区块哈希值”字段进行连接三个区块的链。

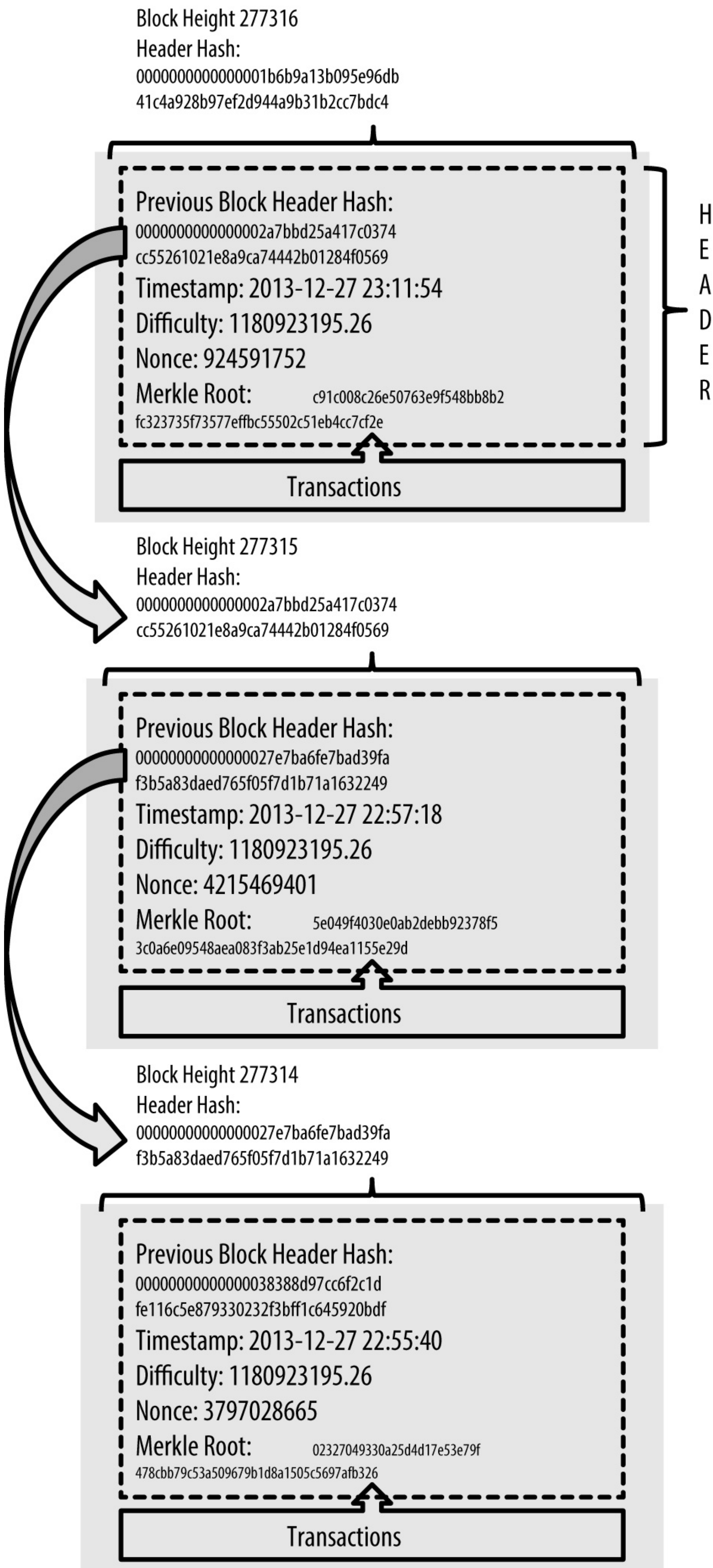

Block Height 277316
Header Hash:
0000000000000001b6b9a13b095e96db
41c4a928b97ef2d944a9b31b2cc7bdc4
Previous Block Header Hash:
0000000000000002a7bbd25a417c0374
cc55261021e8a9ca74442b01284f0569
Timestamp: 2013-12-27 23:11:54
Difficulty: 1180923195.26
Nonce: 924591752
Merkle Root: c91c008c26e50763e9f548bb8b2
fc323735f73577effbc55502c51eb4cc7cf2e
Transactions
HEADER
Block Height 277315
Header Hash:
0000000000000002a7bbd25a417c0374
cc55261021e8a9ca74442b01284f0569
Previous Block Header Hash:
00000000000000027e7ba6fe7bad39fa
f3b5a83daed765f05f7d1b71a1632249
Timestamp: 2013-12-27 22:57:18
Difficulty: 1180923195.26
Nonce: 4215469401
Merkle Root: 5e049f4030e0ab2debb92378f5
3c0a6e09548aea083f3ab25e1d94ea1155e29d
Transactions
Block Height 277314
Header Hash:
00000000000000027e7ba6fe7bad39fa
f3b5a83daed765f05f7d1b71a1632249
Previous Block Header Hash:
00000000000000038388d97cc6f2c1d
fe116c5e879330232f3bff1c645920bdf
Timestamp: 2013-12-27 22:55:40
Difficulty: 1180923195.26
Nonce: 3797028665
Merkle Root: 02327049330a25d4d17e53e79f
478cbb79c53a509679b1d8a1505c5697afb326
Transactions

图9-1通过引用前面的区块头哈希连接成区块链

9.7 Merkle 树

区块链中的每个区块都包含了产生于该区块的所有交易，且以Merkle树表示。

Merkle树是一种哈希二叉树，它是一种用作快速归纳和校验大规模数据完整性的数据结构。这种二叉树包含加密哈希值。术语“树”在计算机学科中常被用来描述一种具有分支的数据结构，但是树常常被倒置显示，“根”在图的上部同时“叶子”在图的下部，你会在后续章节中看到相应的例子。

在比特币网络中，Merkle树被用来归纳一个区块中的所有交易，同时生成整个交易集合的数字指纹，且提供了一种校验区块是否存在某交易的高效途径。生成一棵完整的Merkle树需要递归地对一对节点进行哈希，并将新生成的哈希节点插入到Merkle树中，直到只剩一个哈希节点，该节点就是Merkle树的根。在比特币的Merkle树中两次使用到了SHA256 算法，因此其加密哈希算法也被称为double-SHA256。

当N个数据元素经过加密后插入Merkle树时，你至多计算2*log~2~(N) 次就能检查出任意某数据元素是否在该树中，这使得该数据结构非常高效。

Merkle树是自底向上构建的。在如下的例子中，我们从A、B、C、D四个构成Merkle树树叶的交易开始，如图9-2。

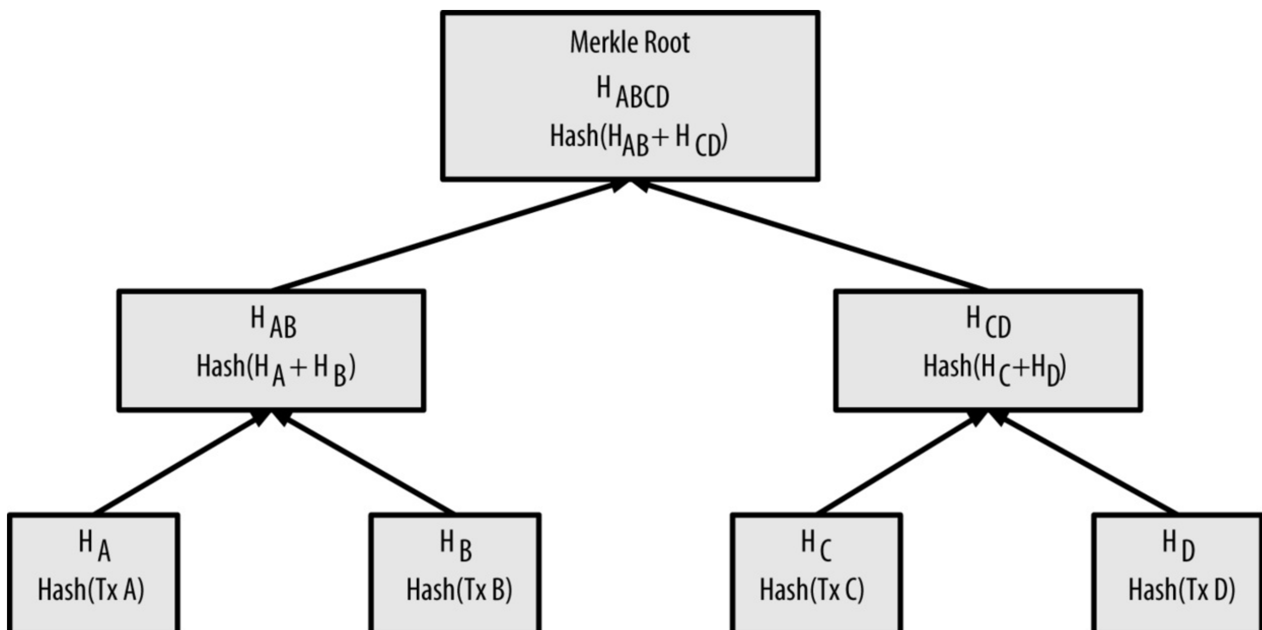


图9-2计算默克树中的节点

所有的交易都并不存储在Merkle树中，而是将数据哈希化，然后将哈希值存储至相应的叶子节点。这些叶子节点分别是H~A~、H~B~、H~C~和H~D~：

HA = SHA256(SHA256(Transaction A))

将相邻两个叶子节点的哈希值串联在一起进行哈希，这对叶子节点随后被归纳为父节点。 例如，为了创建父节点H~AB~，子节点 A和子节点B的两个32字节的哈希值将被串联成64字节的字符串。随后将字符串进行两次哈希来产生父节点的哈希值:

HAB = SHA256(SHA256(H~A~ + H~B~))

继续类似的操作直到只剩下顶部的一个节点，即Merkle根。产生的32字节哈希值存储在区块头，同时归纳了四个交易的所有数据。图9-2展示了如何通过成对节点的哈希值计算Merkle树的根。

因为Merkle树是二叉树，所以它需要偶数个叶子节点。如果仅有奇数个交易需要归纳，那最后的交易就会被复制一份以构成偶数个叶子节点，这种偶数个叶子节点的树也被称为平衡树。如图9-3所示，C节点被复制了一份。

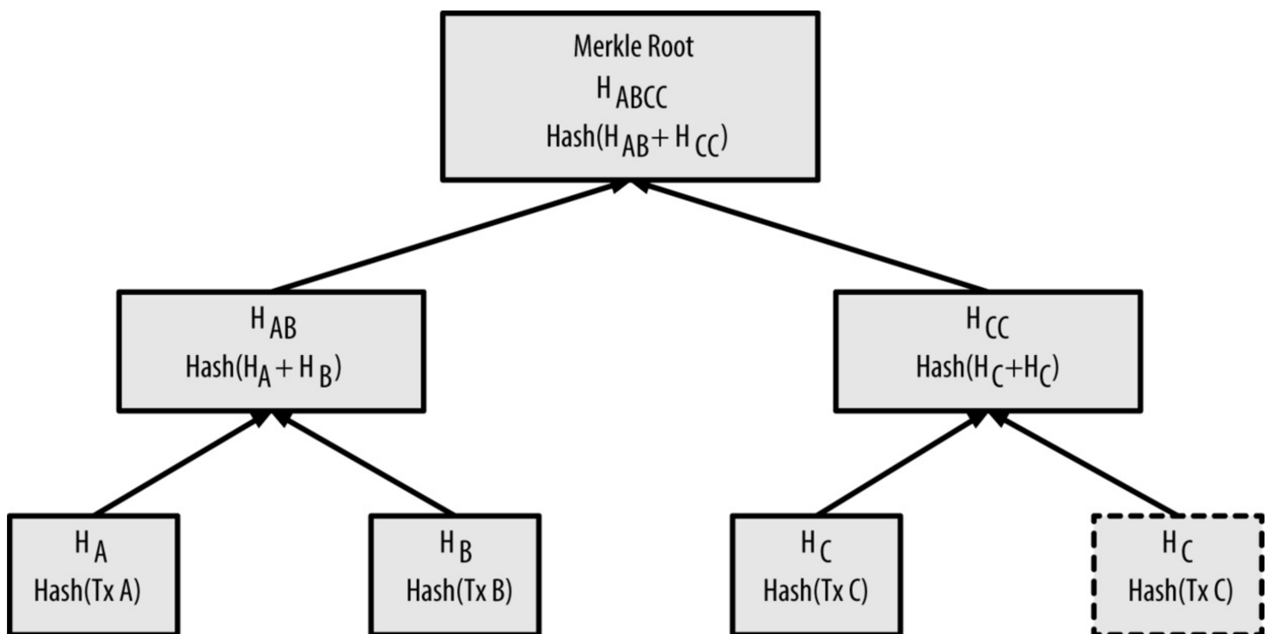


图9-3复制一个数据元素可以实现偶数个数据元素

由四个交易构造Merkle树的方法同样适用于从任意交易数量构造Merkle树。在比特币中，在单个区块中有成百上千的交易是非常普遍的，这些交易都会采用同样的方法归纳起来，产生一个仅仅32字节的数据作为Merkle根。在图9-4中，你会看见一个从16个交易形成的树。需要注意的是，尽管图中的根看起来比所有叶子节点都大，但实际上它们都是32字节的相同大小。无论区块中有一个交易或者有十万个交易，Merkle根总会把所有交易归纳为32字节。

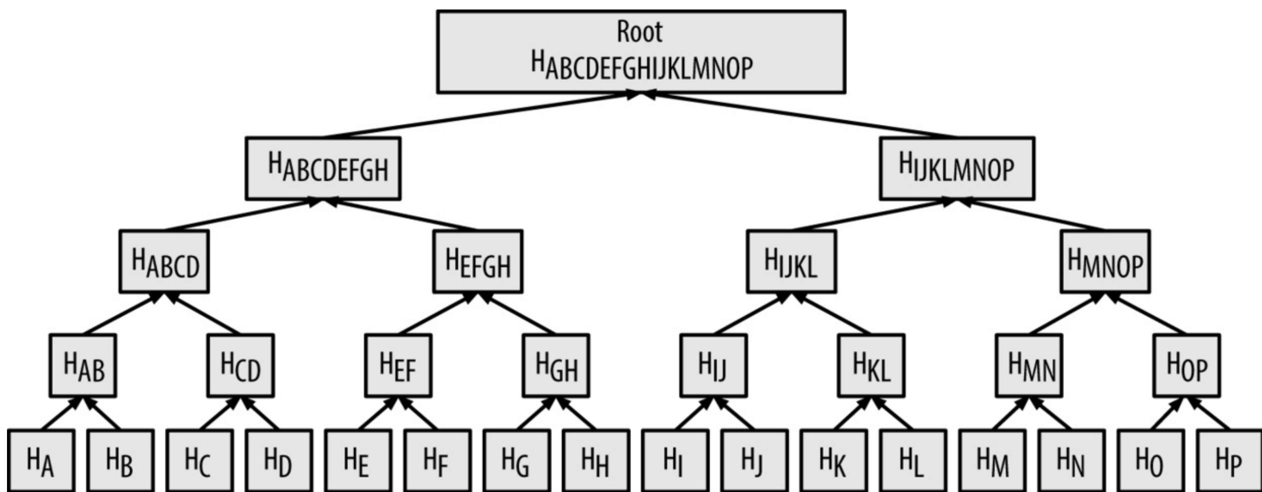


图9-4Merkle树汇总了许多数据元素

为了证明区块中存在某个特定的交易，一个节点只需要计算log~2~(N)个32字节的哈希值，形成一条从特定交易到树根的认证路径或者Merkle路径即可。随着交易数量的急剧增加，这样的计算量就显得异常重要，因为相对于交易数量的增长，以基底为2的交易数量的对数的增长会缓慢许多。这使得比特币节点能够高效地产生一条10或者12个哈希值（320~384字节）的路径，来证明了在一个巨量字节大小的区块中上千交易中的某笔交易的存在。

在图9-5中，一个节点能够通过生成一条仅有4个32字节哈希值长度（总128字节）的Merkle路径，来证明区块中存在一 笔交易K。该路径有4个哈希值（在图9-5中由蓝色标注）H~L~、H~IJ~、H~MNOP~和H~ABCDEFGH~。由这4个哈希值产生的认证路径，再通过计算另外四对哈希值H~KL~、H~IJKL~、H~IJKLMNOP~和Merkle树根（在图中由虚线标注），任何节点都能证明H~K~（在图中由绿色标注）包含在Merkle根中。

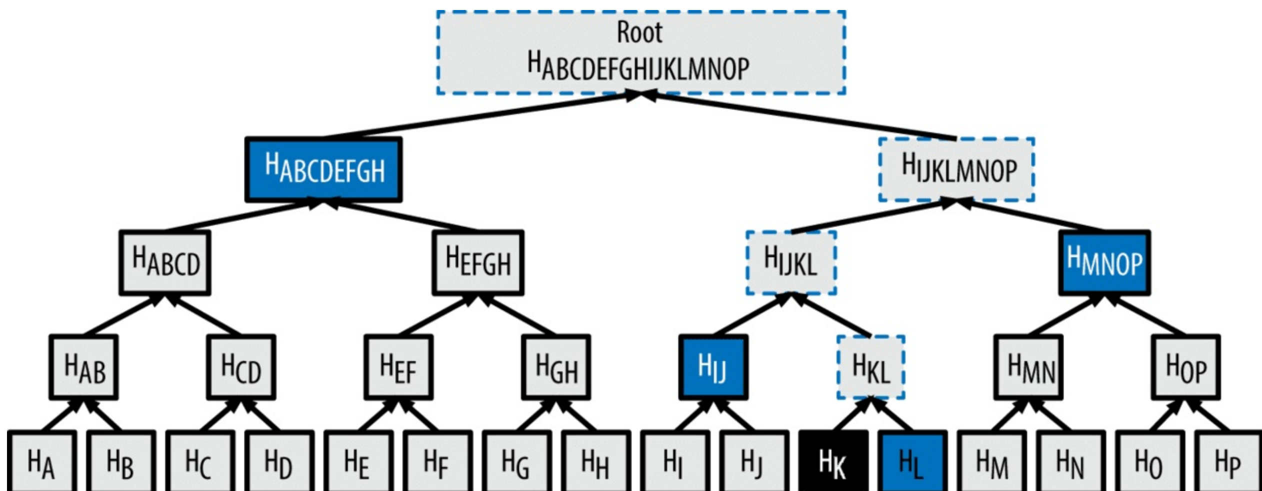


图9-5用于证明包含数据元素的merkle路径

例7-1中的代码借用libbitcoin库中的一些辅助程序，演示了从叶子节点哈希至根创建整个Merkle树的过程。

例9-1 构造Merkle树

link:code/merkle.cpp[]

例9-2展示了编译以及运行上述代码后的结果

\ # Compile the merkle.cpp code

$ g++ -o merkle merkle.cpp

$(pkg-config --cflags --libs libbitcoin)

\ # Run the merkle executable

$ ./merkle

Current merkle hash list:

32650049a0418e4380db0af81788635d8b65424d397170b8499cdc28c4d27006

30861db96905c8dc8b99398ca1cd5bd5b84ac3264a4e1b3e65afa1bcee7540c4

Current merkle hash list:

d47780c084bad3830bcdaf6eace035e4c6cbf646d103795d22104fb105014ba3

Result: d47780c084bad3830bcdaf6eace035e4c6cbf646d103795d22104fb105014ba3

Merkle树的高效随着交易规模的增加而变得异常明显。表9-3展示了为了证明区块中存在某交易而所需转化为Merkle路径的数据量。

Number of transactions	Approx. size of block	Path size (hashes)	Path size (bytes)
16 transactions	4 kilobytes	4 hashes	128 bytes
512 transactions	128 kilobytes	9 hashes	288 bytes
2048 transactions	512 kilobytes	11 hashes	352 bytes
65,535 transactions	16 megabytes	16 hashes	512 bytes

从表中可以看出，当区块大小由16笔交易（4KB）急剧增加至65,535笔交易（16MB）时，为证明交易存在的Merkle路径长度增长极其缓慢，仅仅从128字节到512字节。有了Merkle树，一个节点能够仅下载区块头（80字节/区块），然后通过从一个满节点回溯一条小的Merkle路径就能认证一笔交易的存在，而不需要存储或者传输大量区块链中大多数内容，这些内容可能有几个G的大小。这种不需要维护一条完整的区块链的节点，又被称作简单支付验证（SPV）节点，它不需要下载整个区块而通过Merkle路径去验证交易的存在。

9.8 Merkle树和简单支付验证（SPV）

Merkle树被SPV节点广泛使用。SPV节点不保存所有交易也不会下载整个区块，仅仅保存区块头。它们使用认证路径或者Merkle路径来验证交易存在于区块中，而不必下载区块中所有交易。

例如，一个SPV节点想知道它钱包中某个比特币地址即将到达的支付。该节点会在节点间的通信链接上建立起bloom过滤器，限制只接受含有目标比特币地址的交易。当对等体探测到某交易符合bloom过滤器，它将以Merkleblock消息的形式发送该区块。Merkleblock消息包含区块头和一条连接目标交易与Merkle根的Merkle路径。SPV节点能够使用该路径找到与该交易相关的区块，进而验证对应区块中该交易的有无。SPV节点同时也使用区块头去关联区块和区块链中的其余区块。这两种关联，交易与区块、区块和区块链，就可以证明交易存在于区块链。简而言之，SPV节点会收到少于1KB的有关区块头和Merkle路径的数据，其数据量比一个完整的区块（目前大约有1MB）少了一千多倍。

9.9比特币的测试区块链

你可能会惊讶地发现，有多个比特币区块链。 2009年1月3日由Satoshi Nakamoto创建的“主要”比特币块链，即本章研究的创世区块所在的网络，被称为主干网。另外还有其他用于测试的比特币区块链：现存的有testnet，segnet和regtest。我们依次看看每一个。

9.9.1Testnet——比特币的试验场

Testnet是用于测试的区块链，网络和货币的总称。 testnet是一个功能齐全的在线P2P网络，包括钱包，测试比特币（testnet币），挖矿以及类似主干网的所有其他功能。实际上它和主网只有两个区别：testnet币是毫无价值的，挖掘难度足够低，任何人都可以相对容易地使用testnet币）。

任何打算在比特币主干网上用于生产的软件开发都应该首先在testnet上用测试币进行测试。这样可以保护开发人员免受由于软件错误而导致的金钱损失，也可以保护网络免受由于软件错误导致的意外攻击。

然而，保持测试币的无价值和易挖掘并不容易。尽管有来自开发商的呼吁，但还是有人使用先进的设备（GPU和ASIC）在testnet上挖矿。这就增加了难度，使用CPU挖矿不可能，导致获取测试币非常困难，以致于人们开始赋予其一定价值，所以测试币并不是毫无价值。结果，时不时地testnet必须被报废并重新从创始区块启动，重新进行难度设置。

目前的testnet被称为testnet3，是testnet的第三次迭代，于2011年2月重启，重置了之前的testnet网络的难度。

请记住，testnet3是一个大区块链，在2017年初超过20 GB。完全同步需要一天左右的时间，并占用您的计算机资源。它不像主干网，也不是“轻量级”。运行testnet节点的一个好方法就是将其运行为一个专用的虚拟机镜像（例如，VirtualBox，Docker，Cloud Server等）。

9.9.1.1使用testnet

像几乎所有其他比特币软件一样，Bitcoin Core完全支持在testnet网络运行而不是只能在主干网上运行，还允许您进行测试币挖矿并运行一个testnet全节点。

如果要在testnet上启动Bitcoin Core，而不是在主干网启动，您可以使用testnet开关：

```
$ bitcoind -testnet
```

在日志中，您应该会看到，bitcoind正在默认bitcoind目录的testnet3子目录中构建一个新的区块链：

```
bitcoind: Using data directory /home/username/.bitcoin/testnet3
```

要连接bitcoind，可以使用bitcoin-cli命令行工具，但是要记得切换到testnet模式：

```
$ bitcoin-cli -testnet getinfo
```


```
{
"version": 130200,
"protocolversion": 70015,
"walletversion": 130000,
"balance": 0.00000000,
"blocks": 416,
"timeoffset": 0,
"connections": 3,
"proxy": "",
"difficulty": 1,
"testnet": true,
"keypoololdest": 1484801486,
"keypoolsize": 100,
"paytxfee": 0.00000000,
"relayfee": 0.00001000,
"errors": ""
}
```

您还可以使用getblockchaininfo命令确认testnet3区块链的详细信息和同步进度：

$ bitcoin-cli -testnet getblockchaininfo

{

"chain": "test",

"blocks": 1088,

"headers": 139999,

"bestblockhash":
"0000000063d29909d475a1c4ba26da64b368e56cce5d925097bf3a2084370128",

"difficulty": 1,

"mediantime": 1337966158,

"verificationprogress": 0.001644065914099759,

"chainwork":
"0000000000000000000000000000000000000000000000000000044104410441",

"pruned": false,

"softforks": [

[...]

在testnet3上，你也可以运行使用其他语言和框架实现的全节点来实验和学习，例如btcd（用Go编写）和bcoin（用JavaScript编写）。

在2017年初，testnet3支持主网的所有功能，也包括在主干网络上尚未激活的隔离见证（Segregated Witness（见[segwit]隔离见证章节））。 因此，testnet3也可用于测试隔离见证功能。

9.9.2 Segnet—隔离见证测试网络

2016年，启动了一个特殊用途的测试网络，以帮助开发和测试隔离见证（也称为segwit;见[segwit]）。 该测试区块链称为segnet，可以通过运行Bitcoin Core的特殊版本（分支）来连接。

由于已经将segwit添加到testnet3中，因此后来不再使用segnet来测试segwit功能。

在将来，我们可能会看到其他专门用于测试单个功能或主要架构更改（如segnet）的测试网络区块链。

9.9.3 Regtest--本地区块链

Regtest代表“回归测试”，是一种比特币核心功能，允许您创建本地区块链以进行测试。 与testnet3（它是一个公共和共享的测试区块链）不同，regtest区块链旨在作为本地测试的封闭系统运行。 您从头开始启动regtest区块链，创建一个本地的创世区块。 您可以将其他节点添加到网络中，或者使用单个节点运行它来测试Bitcoin Core软件。

要在regtest模式下启动Bitcoin Core，您可以使用regtest标志：

$ bitcoind -regtest

就像使用testnet一样，Bitcoin Core将在bitcoind默认目录的regtest子目录下初始化一个新的区块链：

bitcoind: Using data directory /home/username/.bitcoin/regtest

要使用命令行工具，还需要指定regtest标志。 我们来试试getblockchaininfo命令来检查regtest区块链：

$ bitcoin-cli -regtest getblockchaininfo

{ "chain": "regtest",

"blocks": 0,

"headers": 0,

"bestblockhash":
"0f9188f13cb7b2c71f2a335e3a4fc328bf5beb436012afca590b1a11466e2206",

"difficulty": 4.656542373906925e-10,

"mediantime": 1296688602,

"verificationprogress": 1,

"chainwork":
"0000000000000000000000000000000000000000000000000000000000000002",

"pruned": false,

[...]

你可以看到，还没有任何区块。 让我们开始挖一些（500块），赚取奖励：

$ bitcoin-cli -regtest generate 500

["7afed70259f22c2bf11e406cb12ed5c0657b6e16a6477a9f8b28e2046b5ba1ca",

"1aca2f154a80a9863a9aac4c72047a6d3f385c4eec5441a4aafa6acaa1dada14",

"4334ecf6fb022f30fbd764c3ee778fabbd53b4a4d1950eae8a91f1f5158ed2d1",

"5f951d34065efeaf64e54e91d00b260294fcdfc7f05dbb5599aec84b957a7766",

"43744b5e77c1dfece9d05ab5f0e6796ebe627303163547e69e27f55d0f2b9353",

[...]

"6c31585a48d4fc2b3fd25521f4515b18aefb59d0def82bd9c2185c4ecb754327"

]

挖掘所有这些块只需要几秒钟，这样就可以很容易地进行测试。 如果您检查您的钱包余额，您将看到您获得了前400个区块的奖励（Coinbase的奖励必须挖满100块之后才能花费）：

$ bitcoin-cli -regtest getbalance

12462.50000000

9.10使用测试块链进行开发

Bitcoin的各种区块链（regtest，segnet，testnet3，以及主干网）为比特币开发提供了一系列测试环境。 无论您是开发比特币核心还是另一个全节点共识客户端，诸如钱包，交易所，电子商务网站等应用程序，甚至开发新颖的智能合同和复杂的脚本等等，请使用测试区块链网进行开发。

您可以使用测试区块链来建立开发管道。 在开发它时，建议在本地测试代码。 一旦您准备好在公共网络上尝试，请切换到testnet，将您的代码暴露在更加动态的环境中，并提供更多样化的代码和应用程序。 最后，一旦您确信您的代码正常工作，请切换到主网以实现生产部署。 当您进行变更，改进，错误修复等操作时，再次启动这个开发管道，首先在regtest上部署每个变更，然后在testnet上进行测试，最后实现生产。

10.1 简介

“挖矿”这个词有点误导。 一般意义的挖矿类似贵金属的提取，更多将人们的注意力集中到创造每个区块中获得的奖励。 虽然挖矿能够获得这种奖励作为激励，但挖矿的主要目的不是这个奖励或者产生新币。 如果您只是把挖矿视为创建新币的过程，则会将比特币系统中的这个手段（激励）作为挖矿过程的目标。 挖矿最重要的作用是巩固了去中心化的清算交易机制，通过这种机制，交易得到验证和清算//清除。 挖矿是使得比特币与众不同的发明，它实现去中心化的安全机制，是P2P数字货币的基础。

挖矿确保了比特币系统安全，并且在没有中央权力机构的情况下实现了全网络范围的共识。新币发行和交易费的奖励是将矿工的行动与网络安全保持一致的激励计划，同时实现了货币发行。

提示：挖矿的目的不是创造新的比特币。 这是激励机制。 挖矿是一种机制，这种机制实现了去中心化的安全。

矿工们验证每笔新的交易并把它们记录在总帐簿上。每10分钟就会有一个新的区块被“挖掘”出来，每个区块里包含着从 上一个区块产生到目前这段时间内发生的所有交易，这些交易被依次添加到区块链中。我们把包含在区块内且被添加到 区块链上的交易称为“确认”（confirmed）交易，交易经过“确认”之后，新的拥有者才能够花费他在交易中得到的比特币。

矿工们在挖矿过程中会得到两种类型的奖励：创建新区块的新币奖励，以及区块中所含交易的交易费。为了得到这些奖 励，矿工们争相完成一种基于加密哈希算法的数学难题，这些难题的答案包括在新区块中，作为矿工的计算工作量的证 明，被称为”“工作量证明”。该算法的竞争机制以及获胜者有权在区块链上进行交易记录的机制，这二者是比特币安全的基石。

新比特币的生成过程被称为挖矿，是因为它的奖励机制被设计为速度递减模式，类似于贵重金属的挖矿过程。比特币的货币是通过挖矿发行的，类似于中央银行通过印刷银行纸币来发行货币。矿工通过创造一个新区块得到的比特币数量大约 每四年（或准确说是每210,000个块）减少一半。开始时为2009年1月每个区块奖励50个比特币，然后到2012年11月减 半为每个区块奖励25个比特币。之后在2016年7月 再次减半为每个新区块奖励12.5个比特币。基于这个公 式，比特币挖矿奖励以指数方式递减，直到2140年。届时所有的比特币（20,999,999,980）全部发行完毕。换句话说 在2140年之后，不会再有新的比特币产生。

矿工们同时也会获取交易费。每笔交易都可能包含一笔交易费，交易费是每笔交易记录的输入和输出的差额。在挖矿过 程中成功“挖出”新区块的矿工可以得到该区块中包含的所有交易“小费”。目前，这笔费用占矿工收入的0.5%或更少，大 部分收益仍来自挖矿所得的比特币奖励。然而随着挖矿奖励的递减，以及每个区块中包含的交易数量增加，交易费在矿 工收益中所占的比重将会逐渐增加。在2140年之后，所有的矿工收益都将由交易费构成。

在本章中，我们先来审视比特币的货币发行机制，然后再来了解挖矿的最重要的功能：支撑比特币安全的去中心化的共识机制。

为了了解挖矿和共识，我们将跟随Alice的交易，以及上海的矿工Jing如何收到并利用挖矿设备将交易加入区块。 然后，我们将继续跟踪区块被挖矿，加入区块链，并通过共识被比特币网络接受。

10.1.1 比特币经济学和货币创造

通过创造出新区块，比特币以一个确定的但不断减慢的速率被铸造出来。大约每十分钟产生一个新区块，每一个新区块 都伴随着一定数量从无到有的全新比特币。每开采210,000个块，大约耗时4年，货币发行速率降低50%。在比特币运 行的第一个四年中，每个区块创造出50个新比特币。

2012年11月，比特币的新发行速度降低到每区块25个比特币。2016年7月，降低到12.5比特币/区块。2020年的某个时候，也就是在区块630,000，它将再次下降至6.25比特币。新币的发行速度会以指数级进行32次“等分”，直到第6,720,000块（大约在2137年开采），达到比特币的最小货币单位1 satoshi。最终，在经过693万个区块之后，所有的共2,099,999,997,690,000聪比特币将全部发行完毕。也就是说，到2140年左右，会存在接近2,100万比特币。在那之后，新的区块不再包含比特币奖励，矿工的收益全部来自交易费。图10-1展示了在发行速度不断降低的情况下，比特币总流通量与时间的关系。

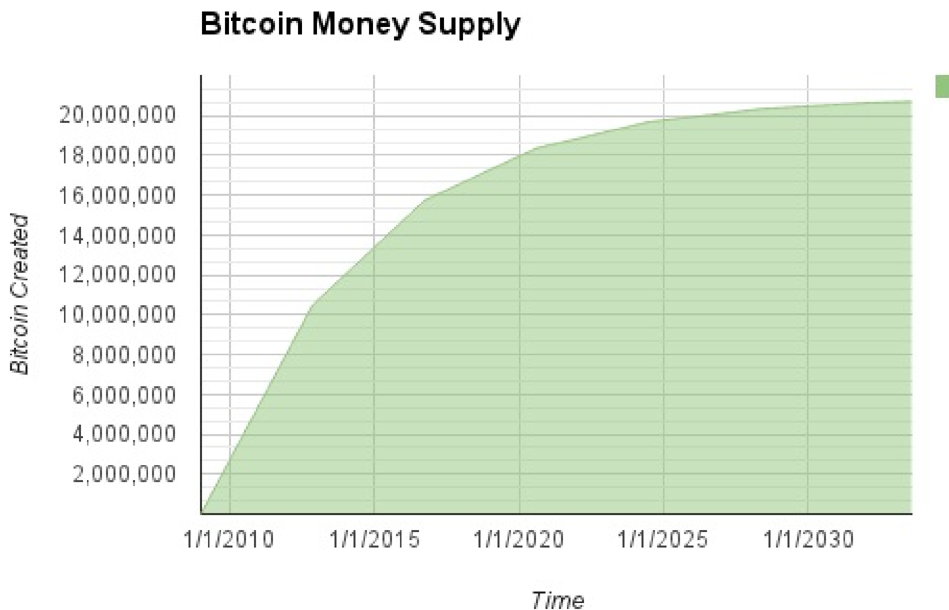


图10-1 比特币发行与时间的关系

注意：比特币挖矿发行的最大数量也就成为挖矿奖励的上限。 在实践中，矿工可能故意挖掘哪些不足全额奖励的区块。 这些块已经开采了，未来可能会有更多被开采，这样导致货币发行总量的下降。

在例10-1的代码展示中，我们计算了比特币的总发行量

例10-1 比特币发行总量的计算脚本

```
link:code/max_money.py[]
```

Running the max_money.py script说明了运行脚本的输出结果

例10-2 显示上述脚本的输出

```
$ python max_money.pyTotal BTC to ever be created: 2099999997690000 Satoshis
```

总量有限并且发行速度递减创造了一种抗通胀的货币供应模式。法币可被中央银行无限制地印刷出来，而比特币永远不会因超额印发而出现通胀。

通货紧缩

最重要并且最有争议的一个结论是一种事先确定的发行速率递减的货币发行模式会导致货币通货紧缩（简称通缩）。通缩是一种由于货币的供应和需求不匹配导致的货币增值的现象。它与通胀相反，价格通缩意味着货币随着时间有越来越强的购买力。

许多经济学家提出通缩经济是一种无论如何都要避免的灾难型经济。因为在快速通缩时期，人们预期着商品价格会下跌，人们将会储存货币，避免花掉它。这种现象充斥了日本经济“失去的十年”，就是因为在需求坍塌之后导致了滞涨状态。

比特币专家们认为通缩本身并不坏。更确切地说，我们将通缩与需求坍塌联系在一起是因为过去出现的一个特例。在法币届，货币是有可能被无限制印刷出来的，除非遇到需求完全崩塌并且毫无发行货币意愿的情形，因此经济很难进入滞涨期。而比特币的通缩并不是需求坍塌引起的，它遵循一种预定且有节制的货币供应模型。

通货紧缩的积极因素当然是与通货膨胀相反。 通货膨胀导致货币缓慢但不可避免的贬值，这是一种隐性税收的形式，惩罚在银行存钱的人从而实现解救债务人（包括政府这个最大的债务人）。 政府控制下的货币容易遭受债务发行的道德风险，之后可能会以牺牲储蓄者为代价，通过贬值来抹去债务。

比特币这种不是因经济快速衰退而引起的通缩，是否会引发其他问题，仍有待观察。

10.2 去中心化共识

在上一章中我们了解了区块链。可以将区块链看作一本记录所有交易的公开总帐簿（列表），比特币网络中的每个参与 者都把能接受区块链，把它看作一本证明所有权的权威记录。

但在不考虑相信任何人的情况下，比特币网络中的所有参与者如何达成对任意一个所有权的共识呢？所有的传统支付系 统都依赖于一个中心认证机构，依靠中心机构提供的结算服务来验证并处理所有的交易。比特币没有中心机构，几乎所 有的完整节点都有一份公共总帐的备份，这份总帐可以被视为权威记录。区块链并不是由一个中心机构创造的，它是由比特币网络中的所有节点各自独立竞争完成的。换句话说比特币网络中的所有节点，依靠着节点间的不稳定的网络连接所传输的信息，最终得出同样的结果并维护了同一个公共总帐。这一章将介绍比特币网络不依靠中心机构而达成共识的机制。

中本聪的主要发明就是这种去中心化的自发共识（emergent consensus）机制。这种自发，是指共识没有明确的完成点，因为共识达成时，没有明确的选举和固定时刻。换句话说，共识是数以千计的独立节点遵守了简单的规则通过异步交互自发形成的产物。所有的比特币属性，包括货币、交易、支付以及不依靠中心机构和信任的安全模型等都依赖于这个发明。

比特币的去中心化共识由所有网络节点的4种独立过程相互作用而产生：

▷ 每个全节点依据综合标准对每个交易进行独立验证

▷ 通过完成工作量证明算法的验算，挖矿节点将交易记录独立打包进新区块

▷ 每个节点独立的对新区块进行校验并组装进区块链

▷ 每个节点对区块链进行独立选择，在工作量证明机制下选择累计工作量最大的区块链。

在接下来的几节中，我们将审视这些过程，了解它们之间如何相互作用并达成全网的自发共识，从而使任意节点组合出 它自己的权威、可信、公开的总帐副本。

10.3 交易的独立校验

在第6章交易中，我们知道了钱包软件通过收集UTXO、提供正确的解锁脚本、构造一个新的支出（支付）给接收者这一系列的方式来创建交易。产生的交易随后将被发送到比特币网络临近的节点，从而使得该交易能够在整个比特币网络中传播。

然而，在交易传递到临近的节点前，每一个收到交易的比特币节点将会首先验证该交易，这将确保只有有效的交易才会 在网络中传播，而无效的交易将会在第一个节点处被废弃。

每一个节点在校验每一笔交易时，都需要对照一个长长的标准列表：

▷交易的语法和数据结构必须正确。

▷输入与输出列表都不能为空。

▷交易的字节大小是小于 MAX_BLOCK_SIZE 的。

▷每一个输出值，以及总量，必须在规定值的范围内 （小于2,100万个币，大于0）。

▷没有哈希等于0，N等于-1的输入（coinbase交易不应当被传递）。

▷nLockTime是小于或等于 INT_MAX 的。或者nLocktime and nSequence的值满足MedianTimePast（译者注：MedianTime是这个块的前面11个块按照block time排序后的中间时间）

▷交易的字节大小是大于或等于100的。

▷交易中的签名数量(SIGOPS)应小于签名操作数量上限。

▷解锁脚本（ scriptSig ）只能够将数字压入栈中，并且锁定脚本（ scriptPubkey ）必须要符合isStandard的格式 （该格式将会拒绝非标准交易）。

▷池中或位于主分支区块中的一个匹配交易必须是存在的。

▷对于每一个输入，引用的输出是必须存在的，并且没有被花费。

▷对于每一个输入，如果引用的输出存在于池中任何别的交易中（译者注：这笔输入引用的输出有人家自己的输入，不是你），该交易将被拒绝。

▷对于每一个输入，在主分支和交易池中寻找引用的输出交易。如果输出交易缺少任何一个输入，该交易将成为一个孤 立的交易。如果与其匹配的交易还没有出现在池中，那么将被加入到孤立交易池中。

▷对于每一个输入，如果引用的输出交易是一个coinbase输出，该输入必须至少获得COINBASE_MATURITY(100)个确认。

▷使用引用的输出交易获得输入值，并检查每一个输入值和总值是否在规定值的范围内 （小于2100万个币，大于0）。

▷如果输入值的总和小于输出值的总和，交易将被中止。

▷如果交易费用太低以至于无法进入一个空的区块，交易将被拒绝。

▷每一个输入的解锁脚本必须依据相应输出的锁定脚本来验证。

这些条件能够在比特币标准客户端下的 AcceptToMemoryPool 、 CheckTransaction 和 CheckInputs 函数中获得更详细的阐述。 请注意，这些条件会随着时间发生变化，为了处理新型拒绝服务攻击，有时候也为交易类型多样化而放宽规则。

在收到交易后，每一个节点都会在全网广播前对这些交易进行独立校验，并以接收时的相应顺序，为有效的新交易建立一个验证池（还未确认），这个池可以叫做交易池，或者memory pool或者mempool。

10.4 挖矿节点

在比特币网络中，一些节点被称为专业节点“矿工”。第1章中，我们介绍了Jing，在中国上海的计算机工程专业学生，他 就是一位矿工。Jing通过矿机挖矿获得比特币，矿机是专门设计用于挖比特币的计算机硬件系统。Jing的这台专业挖矿设备连接着一个运行完整比特币节点的服务器。与Jing不同，一些矿工是在没有完整节点的条件下进行挖矿，正如我们 在“矿池”一节中所述的。与其他任一完整节点相同，Jing的节点在比特币网络中进行接收和传播未确认交易记录。然而，Jing的节点也能够把这些交易记录打包进入一个新区块。

同其他节点一样，Jing的节点时刻监听着传播到比特币网络的新区块。而这些新加入的区块对挖矿节点有着特殊的意 义。矿工间的竞争以新区块的传播而结束，如同宣布谁是最后的赢家。对于矿工们来说，收到一个新区块进行验证意味着别人已经赢了，而自己则输了这场竞争。然而，一轮竞争的结束也代表着下一轮竞争的开始。新区块并不仅仅是象征着竞赛结束的方格旗；它也是下一个区块竞赛的发令枪。

10.5 打包交易至区块

验证交易后，比特币节点会将这些交易添加到自己的内存池中。内存池也称作交易池，用来暂存尚未被加入到区块的交 易记录。与其他节点一样，Jing的节点会收集、验证并传递新的交易。而与其他节点不同的是，Jing的节点会把这些交 易整合到一个候选区块中。

让我们继续跟进，看下Alice从Bob咖啡店购买咖啡时产生的那个区块。Alice的交易在区块277,316。为了演示本章中提到的概念，我们假设这个区块是由Jing的挖矿系统挖出的，并且继续跟进Alice的交易，因 为这个交易已经成为了新区块的一部分。

Jing的挖矿节点维护了一个区块链的本地副本。当Alice买咖啡 的时候，Jing节点的区块链已经收集到了区块277,314，并继续监听着网络上的交易，在尝试挖掘新区块的同时，也监 听着由其他节点发现的区块。当Jing的节点在挖矿时，它从比特币网络收到了区块277,315。这个区块的到来标志着终 结了产出区块277,315竞赛，与此同时也是产出区块277,316竞赛的开始。

在上一个10分钟内，当Jing的节点正在寻找区块277,315的解的同时，它也在收集交易记录为下一个区块做准备。目前 它已经收到了几百笔交易记录，并将它们放进了内存池。直到接收并验证区块277,315后，Jing的节点会检查内存池中 的全部交易，并移除已经在区块277,315中出现过的交易记录，确保任何留在内存池中的交易都是未确认的，等待被记 录到新区块中。

Jing的节点立刻构建一个新的空区块，做为区块277,316的候选区块。称作候选区块是因为它还没有包含有效的工作量 证明，不是一个有效的区块，而只有在矿工成功找到一个工作量证明解之后，这个区块才生效。

现在，Jing的节点从内存池中整合到了全部的交易，新的候选区块包含有418笔交易，总的矿工费为0.09094925个比特 币。你可以通过比特币核心客户端命令行来查看这个区块，如例10-3所示：

例10-3 使用命令行检索区块277,316

```
$ bitcoin-cli get blockhash 277316
0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4

$ bitcoin-cli getblock 0000000000000001b6b9a13b095e96db41c4a928b97ef2d9\44a9b31b2cc7bd
c4
```

```
{
"hash" : "0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4",


"confirmations" : 35561,


"size" : 218629,


"height" : 277316,


"version" : 2,


"merkleroot" : "c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e",


"tx" : [


  "d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f",


  "b268b45c59b39d759614757718b9918caf0ba9d97c56f3b91956ff877c503fbe",


  ... 417 more transactions ...//这一行显示交易数量


 ],


"time" : 1388185914,


"nonce" : 924591752,


"bits" : "1903a30c",
```

```
"difficulty" : 1180923195.25802612,


"chainwork" : "000000000000000000000000000000000000000000000934695e92aaf53afa1a",


"previousblockhash" : "0000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f056
9"
}
```

10.5.1 创币交易

区块中的第一笔交易是笔特殊交易，称为创币交易或者coinbase交易。这个交易是由Jing的节点构造并用来奖励矿工们所做的贡献的。

注意:当块277,316开采时，每个块的奖励是25比特币。 此后，已经过了一个“减半”时期。2016年七月份的奖励为12.5比特币，2020年达到210000区块时，将再次减半。

Jing的节点会创建“向Jing的地址支付25.09094928个比特币”这样一个交易，把生成交易的奖励发送到自己的钱包。Jing挖出区块获得的奖励金额是coinbase奖励（25个全新的比特币）和区块中全部交易矿工费的总和。

如 例10-4所示：coinbase交易

```
$ bitcoin-cli getrawtransaction d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e
7424afba2f 1
```

```
{

"hex" : "01000000010000000000000000000000000000000000000000000000000000000000000000fff
fffff0f03443b0403858402062f503253482fffffffff0110c08d9500000000232102aa970c592640d19de
03ff6f329d6fd2eecb023263b9ba5d1b81c29b523da8b21ac00000000",


"txid" : "d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f",


"version" : 1,


"locktime" : 0,


"vin" : [


  {
```

```
  "coinbase" : "03443b0403858402062f503253482f",

  "sequence" : 4294967295

  }

],

"vout" : [

  {

  "value" : 25.09094928,

  "n" : 0,

  "scriptPubKey" : {

  "asm" : "02aa970c592640d19de03ff6f329d6fd2eecb023263b9ba5d1b81c29b523da8b21OP_CHECKS
IG",

  "hex" : "2102aa970c592640d19de03ff6f329d6fd2eecb023263b9ba5d1b81c29b523da8b21ac",

  "reqSigs" : 1,

  "type" : "pubkey",

  "addresses" : ["1MxTkeEP2PmHSMze5tUZ1hAV3YTKu2Gh1N"

  ]
  }
  }
]
}
```

与常规交易不同，创币交易没有输入，不消耗UTXO。它只包含一个被称作coinbase的输入，仅仅用来创建新的比特 币。创币交易有一个输出，支付到这个矿工的比特币地址。创币交易的输出将这25.09094928个比特币发送到矿工的比 特币地址，如本例所示的1MxTkeEP2PmHSMze5tUZ1hAV3YTKu2Gh1N。

10.5.2 Coinbase奖励与矿工费

为了构造创币交易，Jing的节点需要计算矿工费的总额，将这418个已添加到区块交易的输入和输出分别进行求和，然 后用输入总额减去输出总额得到矿工费总额，公式如下：

```
Total Fees = Sum(Inputs) - Sum(Outputs)
```

在区块277,316中，矿工费的总额是0.09094925个比特币。

紧接着，Jing的节点计算出这个新区块正确的奖励额。奖励额的计算是基于区块高度的，以每个区块50个比特币为开 始，每产生210,000个区块减半一次。这个区块高度是277,316，所以正确的奖励额是25个比特币。

详细的计算过程可以参看比特币核心客户端中的GetBlockValue函数，如例10-5所示：

例10-5 计算区块奖励—函数GetBlockValue, Bitcoin Core Client, main.cpp

```
CAmount GetBlockSubsidy(int nHeight, const Consensus::Params
&
 consensusParams){
int halvings = nHeight / consensusParams.nSubsidyHalvingInterval;
// Force block reward to zero when right shift is undefined.
if (halvings
>
= 64)
    return 0;

CAmount nSubsidy = 50 * COIN;
// Subsidy is cut in half every 210,000 blocks which will occur approximately every 4
years.
nSubsidy
>
>
 = halvings;
return nSubsidy;
}
```

变量Subsidy表示初始奖励额，值为 COIN 常量（100,000,000聪)与50的乘积，也就是说初始奖励额为50亿聪。

紧接着，这个函数用当前区块高度除以减半间隔(SubsidyHalvingInterval 函数)得到减半次数（变量 halvings ）。每 210,000个区块为一个减半间隔，对应本例中的区块277316，所以减半次数为1。

变量 halvings 最大值64，如果超出这个值，代码算得的奖励额为0。

然后，这个函数会使用二进制右移操作将奖励额(变量 nSubsidy)右移一位（等同与除以2），每一轮减半右移一次。在这个例子中，对于区块277,316只需要将值为50亿聪的奖励额右移一次，得到25亿聪，也就是25个比特币的奖励额。之所以采用二进制右移操作，是因为相比于整数或浮点数除法，右移操作的效率更高。

最后，将coinbase奖励额（变量 nSubsidy ）与矿工费(nFee)总额求和，并返回这个值。

注意: 如果Jing的挖矿节点把coinbase交易写入区块，那么如何防止Jing奖励自己100甚至1000比特币？ 答案是，不正确的奖励将被其他人视为无效，浪费了Jing用于工作证明的投入。 只有这个区块被大家认可，Jing才能得到报酬。

10.5.3创币交易的结构

经过计算，Jing的节点构造了一个创币交易，支付给自己25.09094928枚比特币。

例10-4所示，创币交易的结构比较特殊，与一般交易输入需要指定一个先前的UTXO不同，它包含一个“coinbase“输 入。在之前的章节中，我们已经给出了交易输入的结构。现在让我们来比较一下常规交易输入与创币交易输入。表10-1给出了常规交易输入的结构，表10-2给出的是创币交易输入的结构。

表10-1 常规交易输入结构

Size	Field	Description
32 bytes	Transaction Hash	Pointer to the transaction containing the UTXO to be spent
4 bytes	Output Index	The index number of the UTXO to be spent, first one is 0
1–9 bytes (VarInt)	Unlocking-Script Size	Unlocking-Script length in bytes, to follow
Variable	Unlocking-Script	A script that fulfills the conditions of the UTXO locking script
4 bytes	Sequence Number	Currently disabled Tx-replacement feature, set to 0xFFFFFFFF

表10-2，coinbase交易输入结构

Size	Field	Description
32 bytes	Transaction Hash	All bits are zero: Not a transaction hash reference
4 bytes	Output Index	All bits are ones: 0xFFFFFFFF
1–9 bytes (VarInt)	Coinbase Data Size	Length of the coinbase data, from 2 to 100 bytes
Variable	Coinbase Data	Arbitrary data used for extra nonce and mining tags. In v2 blocks; must begin with block height
4 bytes	Sequence Number	Set to 0xFFFFFFFF

在Coinbase交易中，“交易哈希”字段32个字节全部填充0，“交易输出索引”字段全部填充0xFF(十进制的255)，这两个字段的值表示不引用UTXO。“解锁脚本”由coinbase数据代替，数据可以由矿工自定义。

10.5.4 Coinbase数据

创币交易不包含“解锁脚本“(又称作 scriptSig)字段，这个字段被coinbase数据替代，长度最小2字节，最大100字节。除 了开始的几个字节外，矿工可以任意使用coinbase的其他部分，随意填充任何数据。

以创世块为例，中本聪在coinbase中填入了这样的数据“The Times 03/Jan/ 2009 Chancellor on brink of second bailout for banks“(泰晤士报 2009年1月3日 财政大臣将再次对银行施以援手)，表示对日期的证明，同时也表达了对银行系统的不信任。现在，矿工使用coinbase数据实现extra nonce功能，并嵌入字符串来标识挖出它的矿池，这部分内容会在后面的小节描述。

coinbase前几个字节也曾是可以任意填写的，不过在后来的第34号比特币改进提议(BIP34)中规定了版本2的区块（版本字段为2的区块），这个区块的高度必须跟在脚本操作“push“之后，填充在coinbase字段的起始处。

我们以例10-4中的区块277,316为例，coinbase就是交易输入的“解锁脚本“（或scriptSig）字段，这个字段的十六进制值 为03443b0403858402062f503253482f。下面让我们来解码这段数据。

第一个字节是03，脚本执行引擎执行这个指令将后面3个字节压入脚本栈,紧接着的3个字节——0x443b04， 是以小端格式(最低有效字节在先)编码的区块高度。翻转字节序得到0x043b44，表示为十进制是277,316。

紧接着的几个十六进制数（03858402062）用于编码extra nonce(参见"10.11.1 随机值升位方案")，或者一个随机值，从而求解一个适当的工作量证明。

coinbase数据结尾部分(2f503253482f)是ASCII编码字符 /P2SH/，表示挖出这个区块的挖矿节点支持BIP0016所定义的 pay-to-script-hash(P2SH)改进方案。在P2SH功能引入到比特币的时候，曾经有过一场对P2SH不同实现方式的投票， 候选者是BIP0016和BIP0017。支持BIP0016的矿工将/P2SH/放入coinbase数据中，支持BIP0017的矿工将 p2sh/CHV 放入他们的coinbase数据中。最后，BIP0016在选举中胜出，直到现在依然有很多矿工在他们的coinbase中填 入/P2SH/以表示支持这个功能。

10-6使用了libbitcoin库（在之前“其他替代客户端、资料库、工具包”中提到）从创世块中提取coinbase数据，并显示 出中本聪留下的信息。libbitcoin库中自带了一份创世块的静态拷贝，所以这段示例代码可以直接取自库中的创世块数 据。

例10-6 从创世区块中提取coinbase数据

```
link:code/satoshi-words.cpp\[\]
```

例8-7中，我们使用GNU C++编译器编译源代码并运行得到的可执行文件,

例8-7 编译并运行satoshi-words示例代码

```
$ # Compile the code
$  g++ -o satoshi-words satoshi-words.cpp $(pkg-config --cflags --libs libbitcoin)
$ # Run the executable
$ ./satoshi-words
^D��
<
GS
>
^A^DEThe Times 03/Jan/2009 Chancellor on brink of second bailout for banks
```

10.6 构造区块头

为了构造区块头，挖矿节点需要填充六个字段，如表10-3中所示。

表10-3 区块头结构

Size	Field	Description
4 bytes	Version	A version number to track software/protocol upgrades
32 bytes	Previous Block Hash	A reference to the hash of the previous (parent) block in the chain
32 bytes	Merkle Root	A hash of the root of the merkle tree of this block's transactions
4 bytes	Timestamp	The approximate creation time of this block (seconds from Unix Epoch)
4 bytes	Target	The Proof-of-Work algorithm target for this block
4 bytes	Nonce	A counter used for the Proof-of-Work algorithm

在区块277,316被挖出的时候，区块结构中用来表示版本号的字段值为2，长度为4字节，以小端格式编码值为 0x20000000。

接着，挖矿节点需要填充“前区块哈希”，在本例中，这个值为Jing的节点从网络上接收到的区块277,315 的区块头哈希值，它是区块277316候选区块的父区块。区块277,315的区块头哈希值为：

```
0000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569
```

提示：通过选择候选块头中的先前块哈希字段指定的特定父区块，Jing正在通过确认挖矿能力来扩展区块链。 从本质上讲，这是用他的采矿权为最长难度的有效链进行的“投票”。

为了向区块头填充merkle根字段，要将全部的交易组成一个merkle树。创币交易作为区块中的首个交易，后将余下的 418笔交易添至其后，这样区块中的交易一共有419笔。在之前，我们已经见到过“Merkle树”，树中必须有偶数个叶子 节点，所以需要复制最后一个交易作为第420个叶子节点，每个叶子节点是对应交易的哈希值。这些交易的哈希值逐层地、成对地组合，直到最终组合并成一个根节点。merkle数的根节点将全部交易数据摘要为一个32字节长度的值，例10-3中 merkel根的值如下：

```
c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e
```

挖矿节点会继续添加一个4字节的时间戳，以Unix纪元时间编码，即自1970年1月1日0点到当下总共流逝的秒数。本例 中的1388185914对应的时间是2013年12月27日，星期五，UTC/GMT。

接下来，Jing的节点需要填充Target字段（难度目标值），为了使得该区块有效，这个字段定义了所需满足的工作量证明的难度。难度在区块中以“尾数-指数”的格式，编码并存储，这种格式称作target bits（难度位）。这种编码的首字节表示指数，后面的3字节表示尾数(系 数)。以区块277316为例，难度位的值为0x1903a30c，0x19是指数的十六进制格式，后半部0x03a30c是系数。这部分的概念在后面的“难度目标与难度调整”和“难度表示”有详细的解释。

最后一个字段是nonce，初始值为0。

区块头完成全部的字段填充后，挖矿就可以开始进行了。挖矿的目标是找到一个使区块头哈希值小于难度目标的 nonce。挖矿节点通常需要尝试数十亿甚至数万亿个不同的nonce取值，直到找到一个满足条件的nonce值。

10.7 构建区块

既然Jing的节点已经构建了一个候选区块，那么就轮到Jing的矿机对这个新区块进行“挖掘”，求解工作量证明算法以使这个区块有效。从本书中我们已经学习了比特币系统中不同地方用到的哈希加密函数。比特币挖矿过程使用的是 SHA256哈希函数。

用最简单的术语来说，挖矿就是重复计算区块头的哈希值，不断修改该参数，直到与哈希值匹配的一个过程。哈希函数 的结果无法提前得知，也没有能得到一个特定哈希值的模式。哈希函数的这个特性意味着：得到哈希值的唯一方法是不断的尝试，每次随机修改输入，直到出现适当的哈希值。

10.7.1 工作量证明算法

哈希函数输入一个任意长度的数据，输出一个长度固定且绝不雷同的值，可将其视为输入的数字指纹。对于特定输 入，哈希的结果每次都一样，任何人都可以用相同的哈希函数，计算和验证哈希结果。一个加密哈希函数的主要特征就是不同的 输入几乎不可能出现相同的数字指纹。因此，有意的选择一个输入去生成一个想要的哈希值值是几乎不可能的，更别提用随机的方式生成想要的哈希值了。

无论输入的大小是多少，SHA256函数的输出的长度总是256bit。在例10-8中，我们将使用Python解释器来计算语句 "I am Satoshi Nakamoto" 的SHA256的哈希值。

例10-8 SHA256示例

```
$ python
Python 2.7.1

>
>
>
 import hashlib

>
>
>
 print hashlib.sha256("I am Satoshi Nakamoto").hexdigest()
5d7c7ba21cbbcd75d14800b100252d5b428e5b1213d27c385bc141ca6b47989e
```

在例10-8中，
5d7c7ba21cbbcd75d14800b100252d5b428e5b1213d27c385bc141ca6b47989e 是"I am Satoshi Nakamoto"的哈希值。改变原句中的任何一个字母、标点、或增加字母都会产生不同的哈希值。

如果我们改变原句，得到的应该是完全不同的哈希值。 例如，我们在句子末尾加上一个数字，运行例10-9中的Python脚本。例10-9 通过反复修改 nonce 来生成不同哈希值的脚本（SHA256）

```
link:code/hash\_example.py\[\]
```

执行这个脚本就能生成这些只是末尾数字不同的语句的哈希值。例10-10 中显示了我们只是增加了这个数字，却得到了非 常不同的哈希值。

例10-10 通过反复修改nonce 来生成不同哈希值的脚本的输出

```
$ python hash_example.py
I am Satoshi Nakamoto0 =
>
 a80a81401765c8eddee25df36728d732...
I am Satoshi Nakamoto1 =
>
 f7bc9a6304a4647bb41241a677b5345f...
I am Satoshi Nakamoto2 =
>
 ea758a8134b115298a1583ffb80ae629...
I am Satoshi Nakamoto3 =
>
 bfa9779618ff072c903d773de30c99bd...
I am Satoshi Nakamoto4 =
>
 bce8564de9a83c18c31944a66bde992f...
I am Satoshi Nakamoto5 =
>
 eb362c3cf3479be0a97a20163589038e...
I am Satoshi Nakamoto6 =
>
 4a2fd48e3be420d0d28e202360cfbaba...
I am Satoshi Nakamoto7 =
>
 790b5a1349a5f2b909bf74d0d166b17a...
I am Satoshi Nakamoto8 =
>
 702c45e5b15aa54b625d68dd947f1597...
I am Satoshi Nakamoto9 =
>
 7007cf7dd40f5e933cd89fff5b791ff0...
I am Satoshi Nakamoto10 =
>
```

```
 c2f38c81992f4614206a21537bd634a...
I am Satoshi Nakamoto11 =
>
 7045da6ed8a914690f087690e1e8d66...
I am Satoshi Nakamoto12 =
>
 60f01db30c1a0d4cbce2b4b22e88b9b...
I am Satoshi Nakamoto13 =
>
 0ebc56d59a34f5082aaef3d66b37a66...
I am Satoshi Nakamoto14 =
>
 27ead1ca85da66981fd9da01a8c6816...
I am Satoshi Nakamoto15 =
>
 394809fb809c5f83ce97ab554a2812c...
I am Satoshi Nakamoto16 =
>
 8fa4992219df33f50834465d3047429...
I am Satoshi Nakamoto17 =
>
 dca9b8b4f8d8e1521fa4eaa46f4f0cd...
I am Satoshi Nakamoto18 =
>
 9989a401b2a3a318b01e9ca9a22b0f3...
I am Satoshi Nakamoto19 =
>
 cda56022ecb5b67b2bc93a2d764e75f...
```

每个语句都生成了一个完全不同的哈希值。它们看起来是完全随机的，但你在任何计算机上用Python执行上面的脚本都 能重现这些完全相同的哈希值。

类似这样在语句末尾的变化的数字叫做nonce（随机数）。Nonce是用来改变加密函数输出的，在这个示例中改变了这个语句的 SHA256指纹。

为了使这个哈希算法变得富有挑战，我们来设定一个具有任意性的目标：找到一个语句，使之哈希值的十六进制表示以 0开头。幸运的是，这很容易！在例10-10中语句 "I am Satoshi Nakamoto13" 的哈希值是 0ebc56d59a34f5082aaef3d66b37a661696c2b618e62432727216ba9531041a5 ，刚好满足条件。我们得到它用了13次。用概率的角度 来看，如果哈希函数的输出是平均分布的，我们可以期望每16次得到一个以0开头的哈希值（十六进制个位数字为0到 F）。从数字的角度来看，我们要找的是小于 0x1000000000000000000000000000000000000000000000000000000000000000 的哈希值。

我们称这个为Target目标阈值，我们的目的是找到一个小于这个目标的哈希值。如果我们减小这个目标值，那找到一个小于它的哈希值会越来越难。

简单打个比方，想象人们不断扔一对骰子以得到小于一个特定点数的游戏。第一局，目标是12。只要你不扔出两个6， 你就会赢。然后下一局目标为11。玩家只能扔10或更小的点数才能赢，不过也很简单。假如几局之后目标降低为了5。

现在有一半机率以上扔出来的骰子加起来点数会超过5，因此无效。随着目标越来越小，要想赢的话，扔骰子的次数会 指数级的上升。最终当目标为2时（最小可能点数），只有一个人平均扔36次或2%扔的次数中，他才能赢。从一个知道骰子游戏目标为2的观察者的角度来看，如果有人要成功中奖，假设他平均尝试了36次。

换句话说，可以估计从实现目标难度取得成功所需的工作量。 当算法是基于诸如SHA256的确定性函数时，输入本身就成为证据，必须要一定的工作量才能产生低于目标的结果。 因此，称之为工作量证明。

提示：尽管每次尝试产生一个随机的结果，但是任何可能的结果的概率可以预先计算。 因此，指定特定难度的结果构成了具体的工作量证明。

在例10-10中，成功的nonce为13，且这个结果能被所有人独立确认。任何人将13加到语句 "I am Satoshi Nakamoto" 后面再计算哈希值都能确认它比目标值要小。这个正确的结果同时也是工作量证明（Proof of Work），因为它证明我们的确花时间找到了这个nonce。验证这个哈希值只需要一次计算，而我们找到它却花了13次。如果目标值更小（难度更大），那我们需要多得多的哈希计算才能找到合适的nonce，但其他人验证它时只需要一次哈希计算。此外，知道目标值后，任何人都可以用统计学来估算其难度，因此就能知道找到这个nonce需要多少工作。

提示：工作证明必须产生小于目标的哈希值。 更高的目标意味着找到低于目标的哈希是不太困难的。 较低的目标意味着在目标下方找到哈希更难。 目标和难度是成反比。

比特币的工作量证明和例10-10中的挑战非常类似。矿工用一些交易构建一个候选区块。接下来，这个矿工计算这个区块头信息的哈希值，看其是否小于当前目标值。如果这个哈希值不小于目标值，矿工就会修改这个nonce（通常将之加 1）然后再试一次。按当前比特币系统的难度，矿工得试10^15次（10的15次方）才能找到一个合适的nonce使区块头信息哈希值足够小。

例10-11是一个简化很多的工作量证明算法的实现。

例10-11 简化的工作量证明算法

```
link:code/proof-of-work-example.py\[]
```

你可以任意调整难度值（按二进制bit数来设定，即哈希值开头多少个bit必须是0）。然后执行代码，看看在你的计算机 上求解需要多久。在例10-12中，你可以看到该程序在一个普通笔记本电脑上的执行情况。

例10-12 多种难度值的工作量证明算法的运行输出

```
$ python proof-of-work-example.py*
```

```
Difficulty: 1 (0 bits)

[...]

Difficulty: 8 (3 bits)
Starting search...
Success with nonce 9
Hash is 1c1c105e65b47142f028a8f93ddf3dabb9260491bc64474738133ce5256cb3c1
Elapsed Time: 0.0004 seconds
Hashing Power: 25065 hashes per second
Difficulty: 16 (4 bits)
Starting search...
Success with nonce 25
Hash is 0f7becfd3bcd1a82e06663c97176add89e7cae0268de46f94e7e11bc3863e148
Elapsed Time: 0.0005 seconds
Hashing Power: 52507 hashes per second
Difficulty: 32 (5 bits)
Starting search...
Success with nonce 36
Hash is 029ae6e5004302a120630adcbb808452346ab1cf0b94c5189ba8bac1d47e7903
Elapsed Time: 0.0006 seconds
Hashing Power: 58164 hashes per second

[...]

Difficulty: 4194304 (22 bits)
Starting search...
Success with nonce 1759164
Hash is 0000008bb8f0e731f0496b8e530da984e85fb3cd2bd81882fe8ba3610b6cefc3
Elapsed Time: 13.3201 seconds
Hashing Power: 132068 hashes per second
Difficulty: 8388608 (23 bits)
Starting search...
Success with nonce 14214729
Hash is 000001408cf12dbd20fcba6372a223e098d58786c6ff93488a9f74f5df4df0a3
Elapsed Time: 110.1507 seconds
Hashing Power: 129048 hashes per second
Difficulty: 16777216 (24 bits)
Starting search...
Success with nonce 24586379
Hash is 0000002c3d6b370fccd699708d1b7cb4a94388595171366b944d68b2acce8b95
Elapsed Time: 195.2991 seconds
Hashing Power: 125890 hashes per second

[...]

Difficulty: 67108864 (26 bits)
Starting search...
Success with nonce 84561291
```

```
Hash is 0000001f0ea21e676b6dde5ad429b9d131a9f2b000802ab2f169cbca22b1e21a
Elapsed Time: 665.0949 seconds
Hashing Power: 127141 hashes per second
```

你可以看出，随着难度位一位一位地增加，查找正确结果的时间会呈指数级增长。如果你考虑整个256bit数字空间，每 次要求多一个0，你就把哈希查找空间缩减了一半。在例10-12中，为寻找一个nonce使得哈希值开头的26位值为0，一共尝试了8千多万次。即使家用笔记本每秒可以达270,000多次哈希计算，这个查找依然需要10分钟。

在写这本书的时候，比特币网络要寻找区块头信息哈希值小于

```
0000000000000000029AB9000000000000000000000000000000000000000000
```

可以看出，这个目标哈希值开头的0多了很多。这意味 着可接受的哈希值范围大幅缩减，因而找到正确的哈希值更加困难。生成下一个区块需要网络每秒计算1.8 septa-hashes（(thousand billion billion次哈希）。这看起来像是不可能的任务，但幸运的是比特币网络已经拥有3EH/sec的处理能力，平均每10分钟就可以找到一个新区块。

10.7.2 难度表示

在例10-3中，我们在区块中看到难度目标，其被标为"难度位"或简称"bits"。在区块277,316中，它的值为 0x1903a30c。 这个标记的值被存为系数/指数格式，前两位十六进制数字为幂（exponent），接下来得六位为系数（coefficient）。在这个区块里，0x19为幂，而0x03a30c为系数。

计算难度目标的公式为：

```
target = coefficient \* 2^\(8 \* \(exponent – 3\)\)
```

由此公式及难度位的值 0x1903a30c，可得：

```
    target = 0x03a30c * 2^(0x08 * (0x19 - 0x03))^

     =
>
 target = 0x03a30c * 2^(0x08 * 0x16)^

    =
>
 target = 0x03a30c * 2^0xB0^
```

按十进制计算为：

```
=
>
 target = 238,348 * 2^176^

=
>
 target = 22,829,202,948,393,929,850,749,706,076,701,368,331,072,452,018,388,575,715,3
28
```

转化为十六进制后为：

```
=
>
 target = 0x0000000000000003A30C00000000000000000000000000000000000000000000
```

也就是说高度为277,316的有效区块的头信息哈希值是小于这个目标值的。这个数字的二进制表示中前60位都是0。在 这个难度上，一个每秒可以处理1万亿个哈希计算的矿工（1 tera-hash per second 或 1 TH/sec）平均每8,496个区块才能找到一个正确结果，换句话说，平均每59天，才能为某一个区块找到正确的哈希值。

10.7.3 难度目标与难度调整

如前所述，目标决定了难度，进而影响求解工作量证明算法所需要的时间。那么问题来了：为什么这个难度值是可调整 的？由谁来调整？如何调整？

比特币的区块平均每10分钟生成一个。这就是比特币的心跳，是货币发行速率和交易达成速度的基础。不仅是在短期 内，而是在几十年内它都必须要保持恒定。在此期间，计算机性能将飞速提升。此外，参与挖矿的人和计算机也会不断 变化。为了能让新区块的保持10分钟一个的产生速率，挖矿的难度必须根据这些变化进行调整。事实上，难度是一个动 态的参数，会定期调整以达到每10分钟一个新区块的目标。简单地说，难度被设定在，无论挖矿能力如何，新区块产生 速率都保持在10分钟一个。

那么，在一个完全去中心化的网络中，这样的调整是如何做到的呢？难度的调整是在每个完整节点中独立自动发生的。 每2,016个区块中的所有节点都会调整难度。难度的调整公式是由最新2,016个区块的花费时长与20,160分钟（即 这些区块以10分钟一个速率所期望花费的时长）比较得出的。难度是根据实际时长与期望时长的比值进行相应调整的 （或变难或变易）。简单来说，如果网络发现区块产生速率比10分钟要快时会增加难度。如果发现比10分钟慢时则降低 难度。

这个公式可以总结为如下形式：

```
New Difficulty = Old Difficulty \* \(Actual Time of Last 2016 Blocks / 20160 minutes\)
```

例10-13展示了比特币核心客户端中的难度调整代码。

例10-13 工作量证明的难度调整 源文件(pow.cpp文件钟的CalculateNextWorkRequired() 函数)

第43行函数 GetNextWorkRequired()// Limit adjustment step

```
// Limit adjustment step
    int64_t nActualTimespan = pindexLast-
>
GetBlockTime() - nFirstBlockTime;
    LogPrintf("  nActualTimespan = %d  before bounds\n", nActualTimespan);
    if (nActualTimespan
<
 params.nPowTargetTimespan/4)
        nActualTimespan = params.nPowTargetTimespan/4;
    if (nActualTimespan
>
 params.nPowTargetTimespan*4)
        nActualTimespan = params.nPowTargetTimespan*4;

    // Retarget
    const arith_uint256 bnPowLimit = UintToArith256(params.powLimit);
    arith_uint256 bnNew;
    arith_uint256 bnOld;
    bnNew.SetCompact(pindexLast-
>
nBits);
    bnOld = bnNew;
    bnNew *= nActualTimespan;
    bnNew /= params.nPowTargetTimespan;

    if (bnNew
>
 bnPowLimit)
        bnNew = bnPowLimit;
```

注意：虽然目标校准每2,016个块发生，但是由于Bitcoin Core客户端的一个错误，它是基于之前的2,015个块的总时间（不应该是2,016个），导致重定向偏差向较高难度提高0.05%。

参数Interval(2,016区块)和TargetTimespan(1,209,600秒即两周)的定义在文件chainparams.cpp中。

为了防止难度的变化过快，每个周期的调整幅度必须小于一个因子（值为4）。如果要调整的幅度大于4倍，则按4倍调 整。由于在下一个2,016区块的周期不平衡的情况会继续存在，所以进一步的难度调整会在下一周期进行。因此平衡哈 希计算能力和难度的巨大差异有可能需要花费几个2,016区块周期才会完成。

提示：寻找一个比特币区块需要整个网络花费10分钟来处理，每发现2,016个区块时会根据前2,016个区块完成的时间对难度进行调整。

值得注意的是目标难度与交易的数量和金额无关。这意味着哈希算力的强弱，即让比特币更安全的电力投入量，与交易 的数量完全无关。换句话说，当比特币的规模变得更大，使用它的人数更多时，即使哈希算力保持当前的水平，比特币 的安全性也不会受到影响。哈希算力的增加表明更多的人为得到比特币回报而加入了挖矿队伍。只要为了回报，公平正 当地从事挖矿的矿工群体保持足够的哈希算力，"接管"攻击就不会得逞，让比特币的安全无虞。

目标难度和挖矿电力消耗与将比特币兑换成现金以支付这些电力之间的关系密切相关。高性能挖矿系统就是要用当前硅 芯片以最高效的方式将电力转化为哈希算力。挖矿市场的关键因素就是每度电转换为比特币后的价格。因为这决定着挖 矿活动的营利性，也因此刺激着人们选择进入或退出挖矿市场。

10.8 成功构建区块

前面已经看到，Jing的节点创建了一个候选区块，准备拿它来挖矿。Jing有几个安装了ASIC（专用集成电路）的矿机， 上面有成千上万个集成电路可以超高速地并行运行SHA256算法。这些定制的硬件通过USB连接到他的挖矿节点上。接 下来，运行在Jing的桌面电脑上的挖矿节点将区块头信息传送给这些硬件，让它们以每秒亿万次的速度进行nonce测 试。

在对区块277,316的挖矿工作开始大概11分钟后，这些硬件里的其中一个求得了解并发回挖矿节点。当把这个结果放进 区块头时，nonce 4,215,469,401 就会产生一个区块哈希值：

```
0000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569
```

而这个值小于难度目标值：

```
0000000000000003A30C00000000000000000000000000000000000000000000
```

Jing的挖矿节点立刻将这个区块发给它的所有相邻节点。这些节点在接收并验证这个新区块后，也会继续传播此区块。当这个新区块在网络中扩散时，每个节点都会将它作为区块277,316加到自身节点的区块链副本中。当挖矿节点收到并验证了这个新区块后，它们会放弃之前对构建这个相 同高度区块的计算，并立即开始计算区块链中下一个区块的工作。

下节将介绍节点进行区块验证、最长链选择、达成共识，并以此形成一个去中心化区块链的过程。

10.9 校验新区块

比特币共识机制的第三步是通过网络中的每个节点独立校验每个新区块。当新区块在网络中传播时，每一个节点在将它 转发到其节点之前，会进行一系列的测试去验证它。这确保了只有有效的区块会在网络中传播。独立校验还确保了诚实 的矿工生成的区块可以被纳入到区块链中，从而获得奖励。行为不诚实的矿工所产生的区块将被拒绝，这不但使他们失 去了奖励，而且也浪费了本来可以去寻找工作量证明解的机会，因而导致其电费亏损。

当一个节点接收到一个新的区块，它将对照一个长长的标准清单对该区块进行验证，若没有通过验证，这个区块将被拒 绝。这些标准可以在比特币核心客户端的CheckBlock函数和CheckBlockHead函数中获得，

它包括：

▷ 区块的数据结构语法上有效

▷ 区块头的哈希值小于目标难度（确认包含足够的工作量证明）

▷ 区块时间戳早于验证时刻未来两个小时（允许时间错误）

▷ 区块大小在长度限制之内

▷ 第一个交易（且只有第一个）是coinbase交易

▷ 使用检查清单验证区块内的交易并确保它们的有效性

参见之前章节“交易的独立校验”一节已经讨论过这个清单。每一个节点对每一个新区块的独立校验，确保了矿工无法欺诈。在前面的章节中，我们看到了矿工们如何去记录一笔交 易，以获得在此区块中创造的新比特币和交易费。为什么矿工不为他们自己记录一笔交易去获得数以千计的比特币？这 是因为每一个节点根据相同的规则对区块进行校验。一个无效的coinbase交易将使整个区块无效，这将导致该区块被拒 绝，因此，该交易就不会成为总账的一部分。矿工们必须构建一个完美的区块，基于所有节点共享的规则，并且根据正 确工作量证明的解决方案进行挖矿，他们要花费大量的电力挖矿才能做到这一点。如果他们作弊，所有的电力和努力都 会浪费。这就是为什么独立校验是去中心化共识的重要组成部分。

10.10 区块链的组装与选择

比特币去中心化的共识机制的最后一步是将区块集合至有最大工作量证明的链中。一旦一个节点验证了一个新的区块， 它将尝试将新的区块连接到到现存的区块链，将它们组装起来。

节点维护三种区块：第一种是连接到主链上的，第二种是从主链上产生分支的（备用链），最后一种是在已知链中没有 找到已知父区块的。在验证过程中，一旦发现有不符合标准的地方，验证就会失败，这样区块会被节点拒绝，所以也不 会加入到任何一条链中。

任何时候，主链都是累计了最多难度的区块链。在一般情况下，主链也是包含最多区块的那个链，除非有两个等长的链 并且其中一个有更多的工作量证明。主链也会有一些分支，这些分支中的区块与主链上的区块互为“兄弟”区块。这些区 块是有效的，但不是主链的一部分。

保留这些分支的目的是如果在未来的某个时刻它们中的一个延长了并在难度值上超 过了主链，那么后续的区块就会引用它们。在“10.10.1 区块链分叉”，我们将会看到在同样的区块高度，几乎同时挖出区 块时，候选链是如何产生的。

当节点接收到新区块，它会尝试将这个区块插入到现有区块链中。节点会看一下这个区块的“previous block hash”字 段，这个字段是该区块对其父区块的引用。同时，新的节点将尝试在已存在的区块链中找出这个父区块。大多数情况 下，父区块是主块链的“顶点”，这就意味着这个新的区块延长了主链。举个例子，一个新的区块——区块277,316引用 了它的父区块——区块277,315。收到277316区块的大部分节点都已经将277315最为主链的顶端，因此，连接这个新区块并延长区块链。

有时候，新区块所延长的区块链并不是主链，这一点我们将在“10.10.1 区块链分叉”中看到。在这种情况下，节点将新的区块添加到备用链，同时比较备用链与主链的难度。如果备用链比主链积累了更多的难度，节点将收敛于备用链，意味 着节点将选择备用链作为其新的主链，而之前那个老的主链则成为了备用链。如果节点是一个矿工，它将开始构造新的 区块，来延长这个更新更长的区块链。

如果节点收到了一个有效的区块，而在现有的区块链中却未找到它的父区块，那么这个区块被认为是“孤块”。孤块会被 保存在孤块池中，直到它们的父区块被节点收到。一旦收到了父区块并且将其连接到现有区块链上，节点就会将孤块从 孤块池中取出，并且连接到它的父区块，让它作为区块链的一部分。当两个区块在很短的时间间隔内被挖出来，节点有 可能会以相反的顺序接收到它们，这个时候孤块现象就会出现。

选择了最大难度的区块链后，所有的节点最终在全网范围内达成共识。随着更多的工作量证明被添加到链中，链的暂时 性差异最终会得到解决。挖矿节点通过“投票”来选择它们想要延长的区块链，当它们挖出一个新块并且延长了一个链， 新块本身就代表它们的投票。

相互竞争的链之间是存在差异的，下节我们将看到节点是怎样通过独立选择最长难度链来解决这种差异的。

10.10.1 区块链分叉

因为区块链是去中心化的数据结构，所以不同副本之间不能总是保持一致。区块有可能在不同时间到达不同节点，导致节点有不同的区块链全貌。解决的办法是，每一个节点总是选择并尝试延长代表累计了最大工作量证明的区块链，也就 是最长的或最大累计工作的链（greatest cumulative work chain）。节点通过累加链上的每个区块的工作量，得到建立这个链所要付出的工作量证明的总量。只要所有的节点选择最长累计工作的区块链，整个比特币网络最终会收敛到一致的状态。分叉即在不同区块链间发生的临时差异，当更多的区块添加到了某个分叉中，这个问题便会迎刃而解。

提示由于全球网络中的传输延迟，本节中描述的区块链分叉自动会发生。 我们也将在本章稍后再看看故意引起的分叉。

在接下来的几个图表中，我们将通过网络跟踪“fork”事件的进展。 该图是比特币网络的简化表示。 为了便于描述，不同的区块被显示为不同的形状（星形，三角形，倒置三角形，菱形），遍布网络。 网络中的每个圆表示一个节点。

每个节点都有自己的全局区块链视图。 当每个节点从其邻居接收区块时，它会更新其自己的区块链副本，选择最大累积工作链。 为便于描述，每个节点包含一个图形形状，表示它相信的区块处于主链的顶端。 因此，如果在节点里面看到星形，那就意味着该节点认为星形区块处于主链的顶端。

在第一张图（图10-2）中，网络有一个统一的区块链视角，以星形区块为主链的“顶点”。

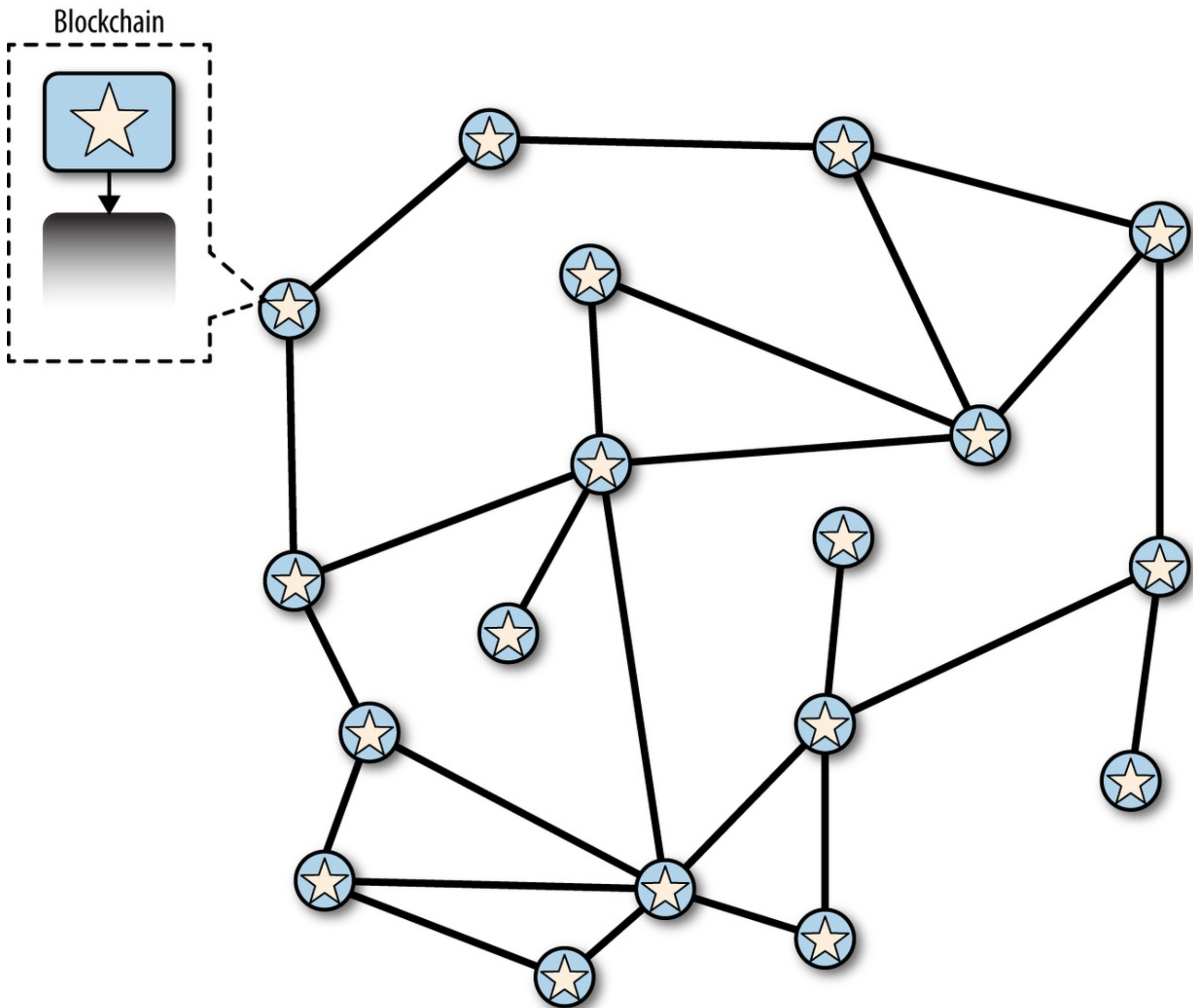


图10-2

当有两个候选区块同时想要延长最长区块链时，分叉事件就会发生。正常情况下，分叉发生在两名矿工在较短的时间 内，各自都算得了工作量证明解的时候。两个矿工在各自的候选区块一发现解，便立即传播自己的“获胜”区块到网络 中，先是传播给邻近的节点而后传播到整个网络。每个收到有效区块的节点都会将其并入并延长区块链。如果该节点在 随后又收到了另一个候选区块，而这个区块又拥有同样父区块，那么节点会将这个区块连接到候选链上。其结果是，一 些节点收到了一个候选区块，而另一些节点收到了另一个候选区块，这时两个

不同版本的区块链就出现了。在图10-3中，我们看到两个矿工（NodeX和NodeY）几乎同时挖到了两个不同的区块。这两个区块是顶点区块——星形区块的子区块，可以 延长这个区块链。为了方便查看，我们把节点X产生的区块标记为三角形，把节点Y生产的区块标记为倒三角形。

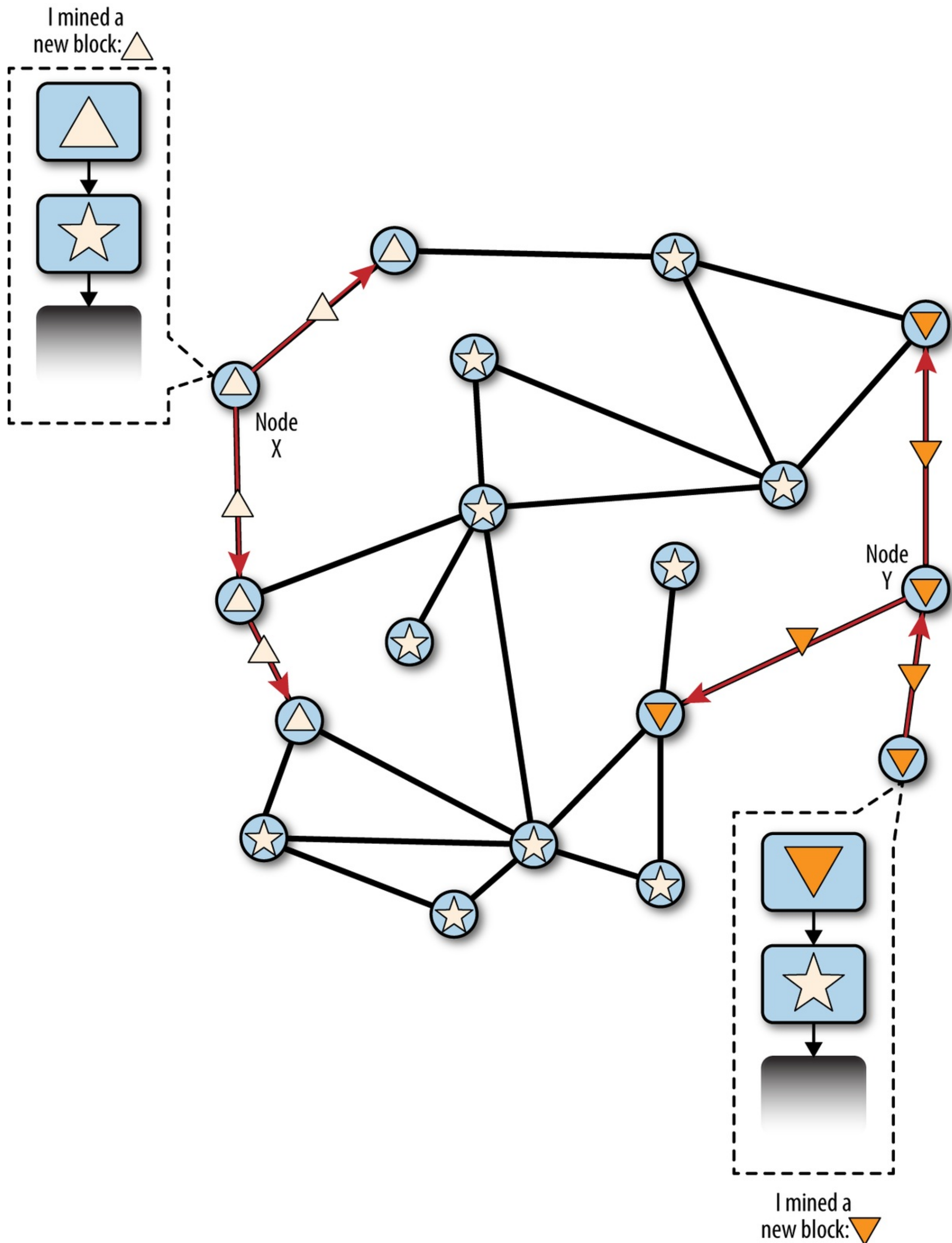


图10-3

例如，我们假设矿工节点X找到扩展区块链工作量证明的解，即三角形区块，构建在星形父区块的顶端。与此同时，同样进行星形区块扩展的节点Y也找到了扩展区块链工作量证明的解，即倒三角形区块作为候选区块。现在有两个可能的块，节点X的三角形区块和节点Y的倒三角形区块，这两个区块都是有效的，均包含有效的工作量证明解并延长同一 个父区块。这个两个区块可能包含了几乎相同的交易，只是在交易的排序上有些许不同。

当两个区块开始在网络传播时，一些节点首先接收到三角形区块，另外一些节点首先接收倒三角形区块。如下图10-4所示，比特币网络上的节点对于区块链的顶点产生了分歧，一派以三角形区块为顶点，而另一派以倒三角形区块为顶点。

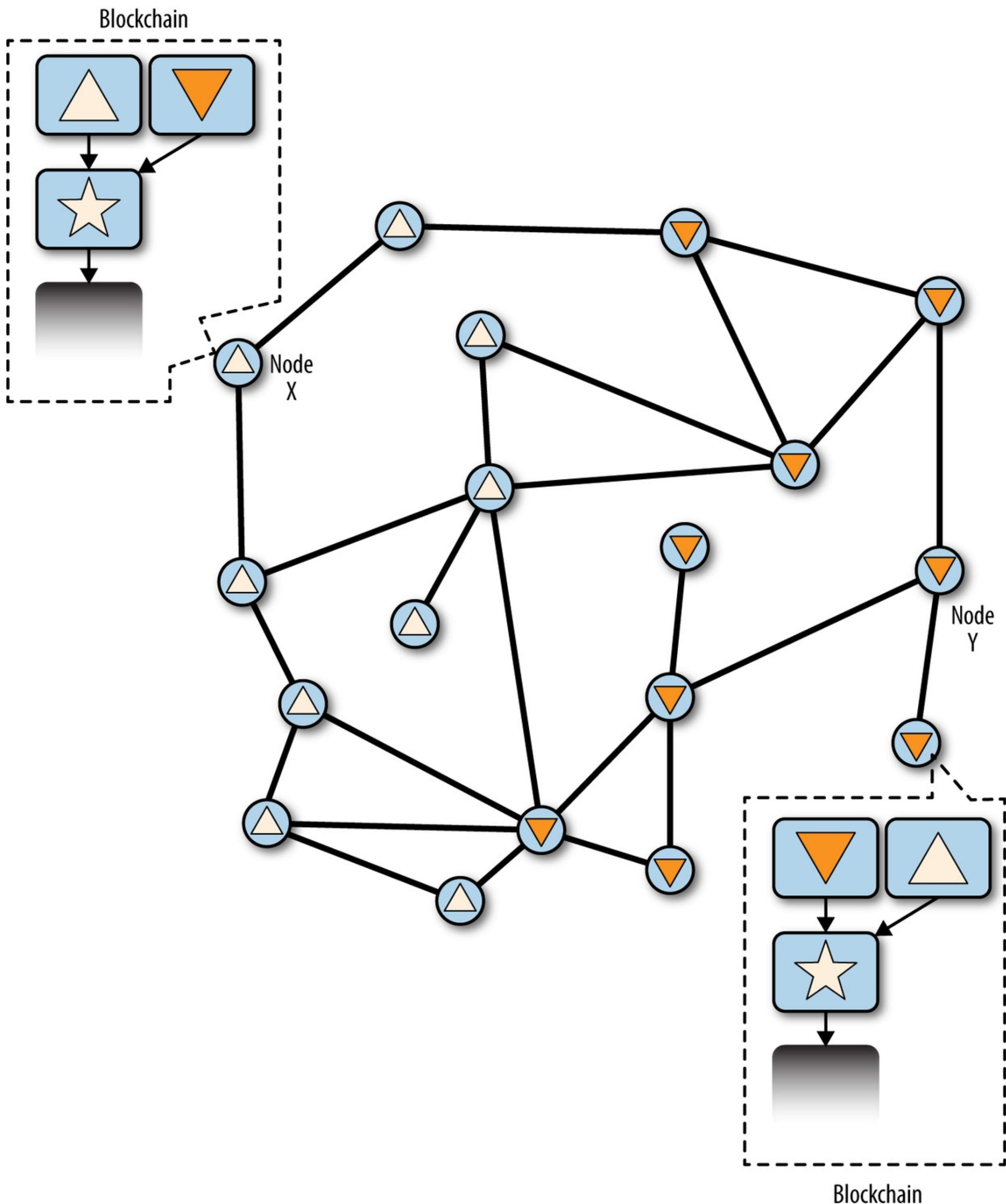


图10-4

在图中，假设节点X首先接收到三角形块，并用它扩展星形链。节点X选择三角形区块为主链。之后，节点X也收到倒三角区块。由于是第二次收到，因此它判定这个倒三角形区块是竞争失败的产物，认为是无效区块。然而，倒三角形的区块不会被丢弃。它被链接到星形链的父区块，并形成备用链。虽然节点X认为自己已经正确选择了获胜链，但是它还会保存“丢失”链，使得“丢失”链如果可能最终“获胜”，它还具有重新打包的所需的信息。

在网络的另一端，节点Y根据自己的视角构建一个区块链。首先获得倒三角形区块，并选择这条链作为“赢家”。当它稍后收到三角形区块时，它也将三角形区块连接到星形链的父区块作为备用链。

双方都是“正确的”或“不正确的”。两者都是自己关于区块链的有效立场。只有事后，才能理解这两个竞争链如何通过额外的工作得到延伸。

节点X阵营的其他节点将立即开始挖掘候选区块，以“三角形”作为扩展区块链的顶端。通过将三角形作为候选区块的父区块，它们用自己的哈希算力进行投票。它们的投票标明支持自己选择的链为主链。

同样，节点Y阵营的其他节点，将开始构建一个以倒三角形作为其父节点的候选节点，扩展它们认为是主链的链。比赛再次开始。分叉问题几乎总是在一个区块内就被解决了。网络中的一部分算力专注于“三角形”区块为父区块，在其之上建立新的区 块；另一部分算力则专注在“倒三角形”区块上。即便算力在这两个阵营中平均分配，也总有一个阵营抢在另一个阵营前发现工作量证明解并将其传播出去。在这个例子中我们可以打个比方，假如工作在“三角形”区块上的矿工找到了一个“菱形”区块 延长了区块链(星形-三角形-菱形)，他们会立刻传播这个新区块，整个网络会都会认为这个区块是有效的，如下图10-5所示。

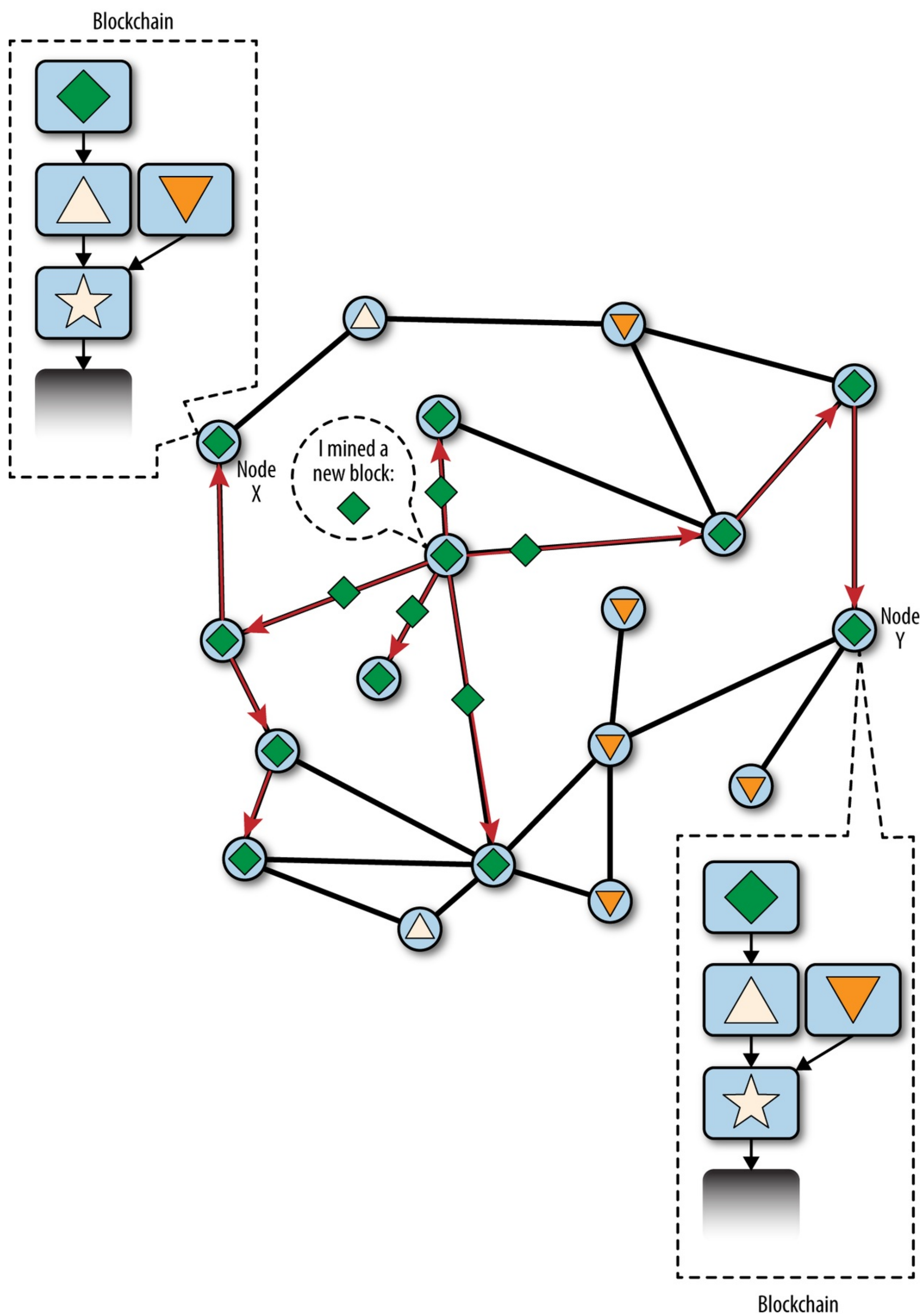


图10-5

选择“三角形”作为上一轮中胜出者的所有节点将简单地将区块链扩展一个块。然而，选择“倒三角”的节点现在将看到两个链：星形-三角形-菱形和星型-到三角形。星形-三角形-菱形这条链现在比其他链条更长（更多累积的工作）。因此，这些节点将星形-三角形-菱形设置为主链，并将星型-倒三角形链变为备用链，如图10-6所示。这是一个链的重新共识，因为这些节点被迫修改他们对块链的立场，把自己纳入更长的链。任何从事延伸星形-倒三角形的矿工现在都将停止这项工作，因为他们的候选人是“孤儿”，因为他们的父母“倒三角形”不再是最长的连锁。“倒三角形”内的交易重新插入到内存池中用来包含在下一个块中，因为它们所在的块不再位于主链中。整个网络重新回到单一链状态，星形-三角形-菱形，“菱形”成为链中的最后一个块。所有矿工立即开始研究以“菱形”为父区块的候选块，以扩展这条星形-三角形-菱形链。

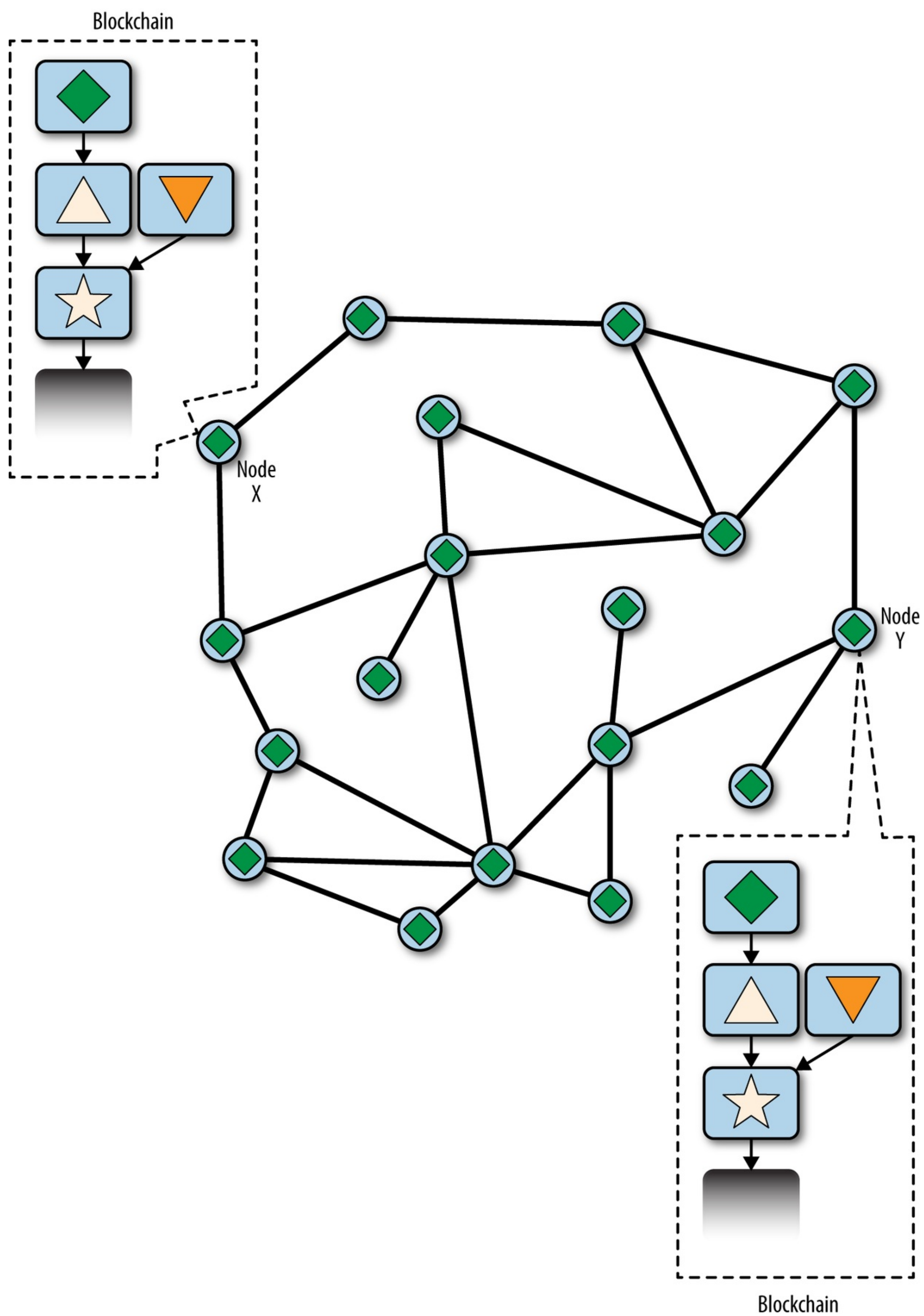


图10-6

从理论上来说，两个区块的分叉是有可能的，这种情况发生在因先前分叉而相互对立起来的矿工，又几乎同时发现了两 个不同区块的解。然而，这种情况发生的几率是很低的。单区块分叉每周都会发生，而双块分叉则非常罕见。比特币将区块间隔设计为10分钟，是在更快速的交易确认和更低的分叉概率间作出的妥协。更短的区块产生间隔会让交 易清算更快地完成，也会导致更加频繁地区块链分叉。与之相对地，更长的间隔会减少分叉数量，却会导致更长的清算时间。

10.11 挖矿和算力竞赛

比特币挖矿是一个极富竞争性的行业。自从比特币存在开始，每年比特币算力都成指数增长。一些年份的增长还体现出技术的变革，比如在2010年和2011年，很多矿工开始从使用CPU升级到使用GPU，进而使用FGPA（现场可编程门阵 列）挖矿。在2013年，ASIC挖矿的引入，把SHA256算法直接固化在挖矿专用的硅芯片上，引起了算力的另一次巨大飞 跃。一台采用这种芯片的矿机可以提供的算力，比2010年比特币网络的整体算力还要大。

下表表示了比特币网络开始运行后最初五年的总算力：

2009

0.5 MH/sec–8 MH/sec (16× growth)

2010

8 MH/sec–116 GH/sec (14,500× growth)

2011

16 GH/sec–9 TH/sec (562× growth)

2012

9 TH/sec–23 TH/sec (2.5× growth)

2013

23 TH/sec–10 PH/sec (450× growth)

2014

10 PH/sec–300 PH/sec (3000× growth)

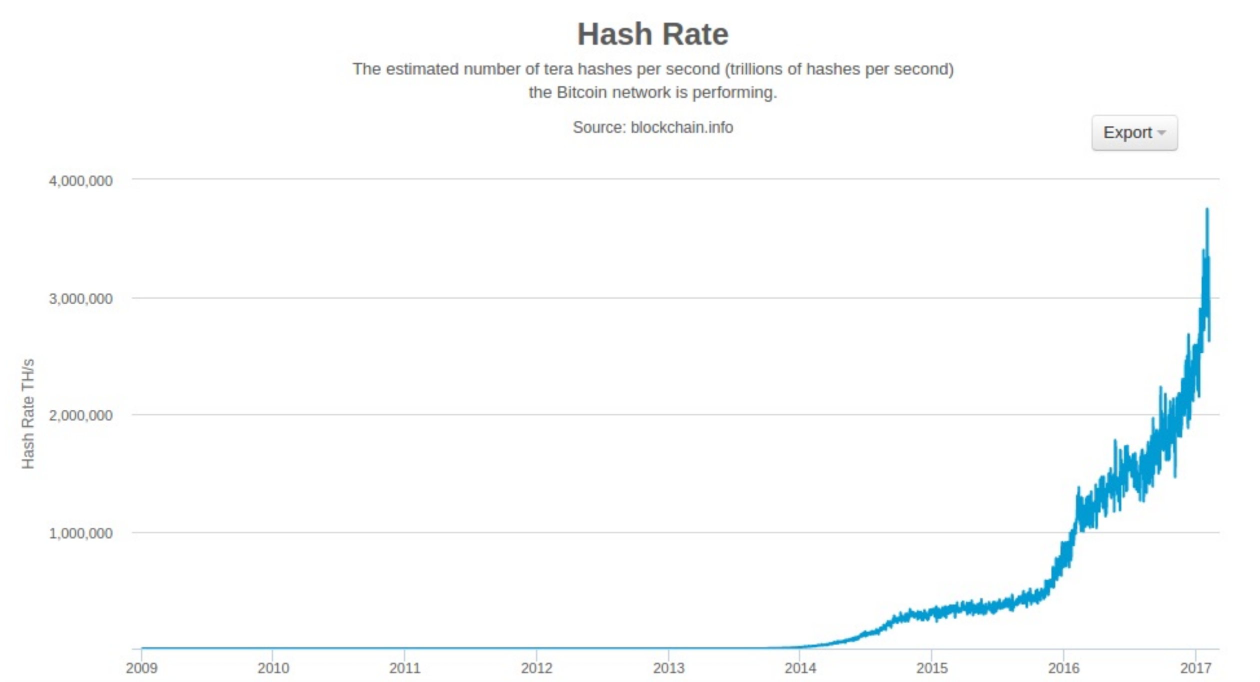
2015

300 PH/sec-800 PH/sec (266× growth)

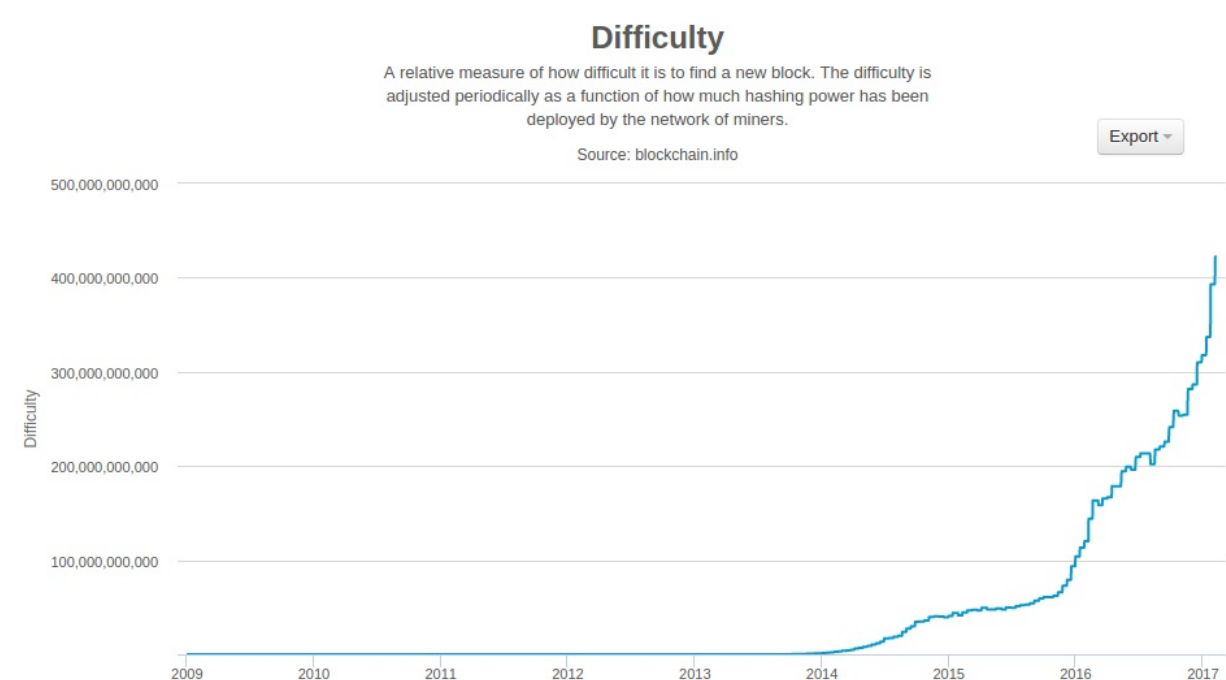
2016

800 PH/sec-2.5 EH/sec (312× growth))

在图10-7的图表中，我们可以看到近两年里，矿业和比特币的成长引起了比特币网络算力的指数增长（每秒网络总算 力）。



随着比特币挖矿算力的爆炸性增长，与之匹配的难度也相应增长。图10-8中的相对难度值显示了当前难度与最小难度 （第一个块的难度）的比例。



近两年，ASIC芯片变得更加密集，特征尺寸接近芯片制造业前沿的16纳米。挖矿的利润率驱动这个行业以比通用计算 更快的速度发展。

目前，ASIC制造商的目标是超越通用CPU芯片制造商，设计特征尺寸为14纳米的芯片。对比特币挖矿而言，已经没有更多飞跃的空间，因为这个行业已经触及了摩尔定律的最前沿。摩尔定律指出计算能力每18个月增加一倍。

尽管如此，随着更高密度的芯片和数据中心的部署竞赛，网络算力继续保持同步的指数增长。现在的竞争已经不再是比较单一芯片的能力，而是一个矿场能塞进多少芯片，并处理好散热和供电问题。

10.11.1 随机值升位方案 the extra nonce solution

2012年以来，比特币挖矿发展出一个解决区块头基本结构限制的方案。在比特币的早期，矿工可以通过遍历随机数(Nonce)获得符合要求的hash来挖出一个块。

难度增长后，矿工经常在尝试了40亿个值后仍然没有出块。然而，这很容 易通过读取块的时间戳并计算经过的时间来解决。因为时间戳是区块头的一部分，它的变化可以让矿工用不同的随机值 再次遍历。当挖矿硬件的速度达到了4GH/秒，这种方法变得越来越困难，因为随机数的取值在一秒内就被用尽了。

当出现ASIC矿机并很快达到了TH/秒的hash速率后，挖矿软件为了找到有效的块，需要更多的空间来储存nonce值。可以把 时间戳延后一点，但将来如果把它移动得太远，会导致区块变为无效。

区块头需要信息来源的一个新的“变革”。解决方案是使用coinbase交易作为额外的随机值来源，因为coinbase脚本可以储存2-100字节的数据，矿工们开始使用这个空间作为额外随机值的来源，允许他们去探索一个大得多的区块头值范围来找到有效的块。这个coinbase交易包含在merkle树中，这意味着任何coinbase脚本的变化将导致Merkle根的变化。

8个字节的额外随机数，加上4个字节的“标准”随机数，允许矿工每秒尝试2^96（8后面跟28个零）种可能性而无需修改时间戳。如果未来矿工穿过了以上所有的可能性，他们还可以通过修改时间戳来解决。同样，coinbase脚本中也有更多额外的空间可以为将来随机数的扩展做准备。

10.11.2 矿池

在这个激烈竞争的环境中，个体矿工独立工作（也就是solo挖矿）没有一点机会。他们找到一个区块以抵消电力和硬件 成本的可能性非常小，以至于可以称得上是赌博，就像是买彩票。就算是最快的消费型ASIC也不能和那些在巨大机房里拥有数万芯片并靠近水电站的商业矿场竞争。

现在矿工们合作组成矿池，汇集数以千计参与者们的算力并分享奖励。通 过参加矿池，矿工们得到整体回报的一小部分，但通常每天都能得到，因而减少了不确定性。

让我们来看一个具体的例子。假设一名矿工已经购买了算力共计14,000GH/S，或14TH/S的设备，在2017年，它的价 值大约是2500美元。

该设备运行功率为1.3千瓦（KW），每日耗电32度，每日平均成本1或2美元。以目前的比特币难度， 该矿工solo方式挖出一个块平均需要4年。如果这个矿工确实在这个时限内挖出一个区块，奖励12.5个比特币，如果 每个比特币价格约为1000美元（译者：这是作者出版此书的

价格，2017年10月22日译者看到的价格是5880美元），可以得到12,500美元的收入。这甚至不能覆盖整个硬件和整个时间段的电力消耗，净亏损约为1,000美元。而且，在4年的时间周期内能否挖出一个块还主要靠矿工的运气。他有可能在4年中得到两个块从而赚到非常大的利润。或者，他可能5年都找不到一个块，从而遭受经济损失。

更糟的是，比特币的工作证明（POW）算法的难度可能在这段时间内显著上升，按照目前算力增长的速度，这意味着矿工在设备被下一代更有效率的矿机取代之前，最多有1年的时间取得成果。如果这个矿工加入矿池，而不是等待4年内可能出现一次的暴利，他每周能赚取大约50-60美元。矿池的常规收入能帮他随时间摊销硬件和电力的成本，并且不用承担巨大的风险。在1-2年后，硬件仍然会过时，风险仍然很高，但在此期间的收入至少是定期的和可靠的。

从财务数据分析，这只有非常低的电力成本（每千瓦不到1美分），非常大的规模时才有意义。矿池通过专用挖矿协议协调成百上千的矿工。

个人矿工在建立矿池账号后，设置他们的矿机连接到矿池服务器。他们的 挖矿设备在挖矿时保持和矿池服务器的连接，和其他矿工同步各自的工作。这样，矿池中的矿工分享挖矿任务，之后分 享奖励。成功出块的奖励支付到矿池的比特币地址，而不是单个矿工的。一旦奖励达到一个特定的阈值，矿池服务器便会定期支 付奖励到矿工的比特币地址。

通常情况下，矿池服务器会为提供矿池服务收取一个百分比的费用。参加矿池的矿工把搜寻候选区块的工作量分割，并根据他们挖矿的贡献赚取“份额”。矿池为赚取“份额”设置了一个低难度 的目标，通常比比特币网络难度低1000倍以上。

当矿池中有人成功挖出一块，矿池获得奖励，并和所有矿工按照他们做 出贡献的“份额”数的比例分配。矿池对任何矿工开放，无论大小、专业或业余。一个矿池的参与者中，有人只有一台小矿机，而有些人有一车库高端挖 矿硬件。有人只用几十度电挖矿，也有人会用一个数据中心消耗兆瓦级的电量。矿池如何衡量每个人的贡献，既能公平 分配奖励，又避免作弊的可能？答案是在设置一个较低难度的前提下，使用比特币的工作量证明算法来衡量每个矿工的 贡献。

因此，即使是池中最小的矿工也经常能分得奖励，这足以激励他们为矿池做出贡献。通过设置一个较低的取得份 额的难度，矿池可以计量出每个矿工完成的工作量。每当矿工发现一个小于矿池难度的区块头hash，就证明了它已经完 成了寻找结果所需的hash计算。更重要的是，这些为取得份额贡献而做的工作，能以一个统计学上可衡量的方法，整体 寻找一个比特币网络的目标散列值。成千上万的矿工尝试较小区间的hash值，最终可以找到符合比特币网络要求的结 果。

让我们回到骰子游戏的比喻。如果骰子玩家的目标是扔骰子结果都小于4（整体网络难度），一个矿池可以设置一个更 容易的目标，统计有多少次池中的玩家扔出的结果小于8。当池中的玩家扔出的结果小于8（矿池份额目标），他们得到份额，但他们没有赢得游戏，因为没有完成游戏目标（小于4）。但池中的玩家会更经常的达到较容易的矿池份额目 标，规律地赚取他们的份额，尽管他们没有完成更难的赢得比赛的目标。时不时地，池中的一个成员有可能会扔出一个小于4的结果，矿池获胜。然后，收益可以在池中玩家获得的份额基础上分配。

尽管目标设置为8或更少并没有赢得游戏，但是这是一个衡量玩家们扔出的点数的公平方法，同时它偶尔会产生一个小于4的结果。同样的，一个矿池会将矿池难度设置在保证一个单独的矿工能够频繁地找到一个符合矿池难度的区块头hash来赢取份 额。时不时的，某次尝试会产生一个符合比特币网络目标的区块头hash，产生一个有效块，然后整个矿池获胜。

10.11.2.1 托管矿池

大部分矿池是“托管的”，意思是有一个公司或者个人经营一个矿池服务器。矿池服务器的所有者叫矿池管理员，同时他 从矿工的收入中收取一个百分比的费用。矿池服务器运行专业软件以及协调池中矿工们活动的矿池采矿协议。矿池服务器同时也连接到一个或更多比特币完全节 点并直接访问一个块链数据库的完整副本。这使得矿池服务器可以代替矿池中的矿工验证区块和交易，缓解他们运行一 个完整节点的负担。

对于池中的矿工，这是一个重要的考量，因为一个完整节点要求一个拥有最少100-150GB的永久储存空间（磁盘）和最少2GB到4GB内存（RAM）的专用计算机。

此外，运行一个完整节点的比特币软件需要监控、维护和频繁升级。由于缺乏维护或资源导致的任何宕机都会伤害到矿工的利润。对于很多矿工来说，不需要跑一个完整节点就能采矿，也是加入托管矿池的一大好处。

矿工连接到矿池服务器使用一个采矿协议比如Stratum(STM)或者 GetBlockTemplate(GBT)。一个旧标准GetWork(GWK)自从2012年底已经基本上过时了，因为它不支持在hash速度超过4GH/S时采矿。STM和GBT协议都创建包含候选区块头模板的区块模板。矿池服务器通过打包交易，添加coinbase交易（和额外的随机值空间），计算MERKLE根， 并连接到上一个块hash来建立一个候选区块。这个候选区块的头部作为模板分发给每个矿工。矿工用这个区块模板在低于比特币网络的难度下采矿，并发送成功的结果返回矿池服务器赚取份额。

10.11.2.2 P2P矿池

托管矿池存在管理人作弊的可能，管理人可以利用矿池进行双重支付或使区块无效。（参见“10.12 共识攻击”） 此外，中 心化的矿池服务器代表着单点故障。如果因为拒绝服务攻击服务器挂了或者被减慢，池中矿工就不能采矿。

在2011年， 为了解决由中心化造成的这些问题，提出和实施了一个新的矿池挖矿方法。P2Pool是一个点对点的矿池，没有中心管理 人。P2Pool通过将矿池服务器的功能去中心化，实现一个并行的类似区块链的系统，名叫份额链（share chain）。

一个份额链是一个难度低 于比特币区块链的区块链系统。份额链允许池中矿工在一个去中心化的池中合作，以每30秒一个份额区块的速度在份额 链上采矿，并获得份额。份额链上的区块记录了贡献工作的矿工的份额，并且继承了之前份额区块上的份额记录。当一 个份额区块上还实现了比特币网络的难度目标时，它将被广播并包含到比特币的区块链上，并奖励所有已经在份额链区 块中取得份额的池中矿工。

本质上说，比起用一个矿池服务器记录矿工的份额和奖励，份额链允许所有矿工通过类似比特币区块链系统的去中心化的共识机制跟踪所有份额。P2Pool采矿方式比在矿池中采矿要复杂的多，因为它要求矿工运行空间、内存、带宽充足的专用计算机来支持一个比特 币的完整节点和P2Pool节点软件。P2Pool矿工连接他们的采矿硬件到本地P2Pool节点，它通过发送区块模板到矿机来 模拟一个矿池服务器的功能。在P2Pool中，单独的矿工创建自己的候选区块，打包交易，非常类似于solo矿工，但是他 们在份额链上合作采矿。

P2Pool是一种比单独挖矿有更细粒度收入优势的混合方法。但是不需要像托管矿池那样给管理 人太多权力。即使P2Pool减少了采矿池运营商的中心化程度，但也可能容易受到份额链本身的51%的攻击。 P2Pool的广泛采用并不能解决比特币本身的51%攻击问题。 相反，作为多样化采矿生态系统的一部分，P2Pool整体使得比特币更加强大。

10.12 共识攻击

比特币的共识机制指的是，被矿工（或矿池）试图使用自己的算力实行欺骗或破坏的难度很大，至少理论上是这样。就像我们前面讲的，比特币的共识机制依赖于这样一个前提，那就是绝大多数的矿工，出于自己利益最大化的考虑，都会 通过诚实地挖矿来维持整个比特币系统。然而，当一个或者一群拥有了整个系统中大量算力的矿工出现之后，他们就可 以通过攻击比特币的共识机制来达到破坏比特币网络的安全性和可靠性的目的。

值得注意的是，共识攻击只能影响整个区块链未来的共识，或者说，最多能影响不久的过去几个区块的共识（最多影响 过去10个块）。而且随着时间的推移，整个比特币块链被篡改的可能性越来越低。

理论上，一个区块链分叉可以变得很 长，但实际上，要想实现一个非常长的区块链分叉需要的算力非常非常大，随着整个比特币区块链逐渐增长，过去的区 块基本可以认为是无法被分叉篡改的。同时，共识攻击也不会影响用户的私钥以及加密算法（ECDSA）。

共识攻击也 不能从其他的钱包那里偷到比特币、不签名地支付比特币、重新分配比特币、改变过去的交易或者改变比特币持有纪 录。共识攻击能够造成的唯一影响是影响最近的区块（最多10个）并且通过拒绝服务来影响未来区块的生成。共识攻击的一个典型场景就是“51%攻击”。想象这么一个场景，一群矿工控制了整个比特币网络51%的算力，他们联合 起来打算攻击整个比特币系统。由于这群矿工可以生成绝大多数的块，他们就可以通过故意制造块链分叉来实现“双重支 付”或者通过拒绝服务的方式来阻止特定的交易或者攻击特定的钱包地址。

区块链分叉/双重支付攻击指的是攻击者通过 不承认最近的某个交易，并在这个交易之前重构新的块，从而生成新的分叉，继而实现双重支付。有了充足算力的保 证，一个攻击者可以一次性篡改最近的6个或者更多的区块，从而使得这些区块包含的本应无法篡改的交易消失。值得注意的是，双重支付只能在攻击者拥有的钱包所发生的交易上进行，因为只有钱包的拥有

者才能生成一个合法的签名用 于双重支付交易。攻击者在自己的交易上进行双重支付攻击，如果可以通过使交易无效而实现对于不可逆转的购买行为不予付款， 这种攻击就是有利可图的。

让我们看一个“51%攻击”的实际案例吧。在第1章我们讲到，Alice 和 Bob 之间使用比特币完成了一杯咖啡的交易。咖啡 店老板 Bob 愿意在 Alice 给自己的转账交易确认数为零的时候就向其提供咖啡，这是因为这种小额交易遭遇“51%攻 击”的风险和顾客购物的即时性（Alice 能立即拿到咖啡）比起来，显得微不足道。这就和大部分的咖啡店对低于25美元 的信用卡消费不会费时费力地向顾客索要签名是一样的，因为和顾客有可能撤销这笔信用卡支付的风险比起来，向用户 索要信用卡签名的成本更高。

相应的，使用比特币支付的大额交易被双重支付的风险就高得多了，因为买家（攻击者） 可以通过在全网广播一个和眞实交易的UTXO一样的伪造交易，以达到取消眞实交易的目的。双重支付可以有两种方 式：要么是在交易被确认之前，要么攻击者通过块链分叉来完成。进行51%攻击的人，可以取消在旧分叉上的交易记 录，然后在新分叉上重新生成一个同样金额的交易，从而实现双重支付。

再举个例子：攻击者Mallory在Carol的画廊买了描绘伟大的中本聪的三联组画（The Great Fire），Mallory通过转账价值25万美金的比特币 与Carol进行交易。在等到一个而不是六个交易确认之后，Carol放心地将这幅组画包好，交给了Mallory。这时，Mallory 的一个同伙，一个拥有大量算力的矿池的人Paul，在这笔交易写进区块链的时候，开始了51%攻击。

首先，Paul利用自己矿池的算力重新计算包含这笔交易的块，并且在新块里将原来的交易替换成了另外一笔交易（比如直接转给了Mallory 的另一个钱包而不是Carol的），从而实现了“双重支付”。这笔“双重支付”交易使用了跟原有交易一致的UTXO，但收款人被替换成了Mallory的钱包地址。

然后，Paul利用矿池在伪造的块的基础上，又计算出一个更新的块，这样，包含这 笔“双重支付”交易的块链比原有的块链高出了一个块。到此，高度更高的分叉区块链取代了原有的区块链，“双重支付”交 易取代了原来给Carol的交易，Carol既没有收到价值25万美金的比特币，原本拥有的三幅价值连城的画也被Mallory白白 拿走了。

在整个过程中，Paul矿池里的其他矿工可能自始至终都没有觉察到这笔“双重支付”交易有什么异样，因为挖矿程序都是自动在运行，并且不会时时监控每一个区块中的每一笔交易。

为了避免这类攻击，售卖大宗商品的商家应该在交易得到全网的6个确认之后再交付商品。或者，商家应该使用第三方 的多方签名的账户进行交易，并且也要等到交易账户获得全网多个确认之后再交付商品。一条交易的确认数越多，越难 被攻击者通过51%攻击篡改。

对于大宗商品的交易，即使在付款24小时之后再发货，对买卖双方来说使用比特币支付也 是方便并且有效率的。而24小时之后，这笔交易的全网确认数将达到至少144个（能有效降低被51%攻击的可能性）。共识攻击中除了“双重支付”攻击，还有一种攻击场景就是拒绝对某个特定的比特币地址提供服务。一个拥有了系统中绝 大多数算力的攻击者，可以轻易地忽略某一笔特定的交易。如果这笔交易存在于另一个矿工所产生的区块中，该攻击者 可以故意分叉，

然后重新产生这个区块，并且把想忽略的交易从这个区块中移除。这种攻击造成的结果就是，只要这名 攻击者拥有系统中的绝大多数算力，那么他就可以持续地干预某一个或某一批特定钱包地址产生的所有交易，从而达到 拒绝为这些地址服务的目的。

需要注意的是，51%攻击并不是像它的命名里说的那样，攻击者需要至少51%的算力才能发起，实际上，即使其拥有不 到51%的系统算力，依然可以尝试发起这种攻击。之所以命名为51%攻击，只是因为在攻击者的算力达到51%这个阈值 的时候，其发起的攻击尝试几乎肯定会成功。

本质上来看，共识攻击，就像是系统中所有矿工的算力被分成了两组，一 组为诚实算力，一组为攻击者算力，两组人都在争先恐后地计算块链上的新块，只是攻击者算力算出来的是精心构造 的、包含或者剔除了某些交易的块。因此，攻击者拥有的算力越少，在这场决逐中获胜的可能性就越小。

从另一个角度 讲，一个攻击者拥有的算力越多，其故意创造的分叉块链就可能越长，可能被篡改的最近的块或者或者受其控制的未来 的块就会越多。一些安全研究组织利用统计模型得出的结论是，算力达到全网的30%就足以发动51%攻击了。全网算力的急剧增长已经使得比特币系统不再可能被某一个矿工攻击，因为一个矿工已经不可能占据全网哪怕的1%算 力。

但是中心化控制的矿池则引入了矿池操作者出于利益而施行攻击的风险。矿池操作者控制了候选块的生成，同时也 控制哪些交易会被放到新生成的块中。这样一来，矿池操作者就拥有了剔除特定交易或者双重支付的权力。如果这种权 利被矿池操作者以微妙而有节制的方式滥用的话，那么矿池操作者就可以在不为人知的情况下发动共识攻击并获益。但是，并不是所有的攻击者都是为了利益。

一个可能的场景就是，攻击者仅仅是为了破坏整个比特币系统而发动攻击， 而不是为了利益。这种意在破坏比特币系统的攻击者需要巨大的投入和精心的计划，因此可以想象，这种攻击很有可能 来自政府资助的组织。同样的，这类攻击者或许也会购买矿机，运营矿池，通过滥用矿池操作者的上述权力来施行拒绝 服务等共识攻击。但是，随着比特币网络的算力呈几何级数快速增长，上述这些理论上可行的攻击场景，实际操作起来 已经越来越困难。

近期比特币系统的一些升级，比如旨在进一步将挖矿控制去中心化的P2Pool挖矿协议，也都正在让这 些理论上可行的攻击变得越来越困难。毫无疑问，一次严重的共识攻击事件势必会降低人们对比特币系统的信心，进而可能导致比特币价格的跳水。然而，比 特币系统和相关软件也一直在持续改进，所以比特币社区也势必会对任何一次共识攻击快速做出响应，以使整个比特币 系统比以往更加稳健和可靠。

10.13改变共识规则

共识规则确定交易和块的有效性。 这些规则是所有比特币节点之间协作的基础，并且负责将所有不同角色的本地视角融合到整个网络中的单一一致的区块链中。虽然共识规则在短期内是不变的，并且在所有节点之间必须一致，但长期来看它们并不总是不变的。 为了演进和开

发比特币系统，规则必须随时改变以适应新功能，改进或修复错误。 然而，与传统软件开发不同，升级到共识系统要困难得多，需要所有参与者之间的协调。

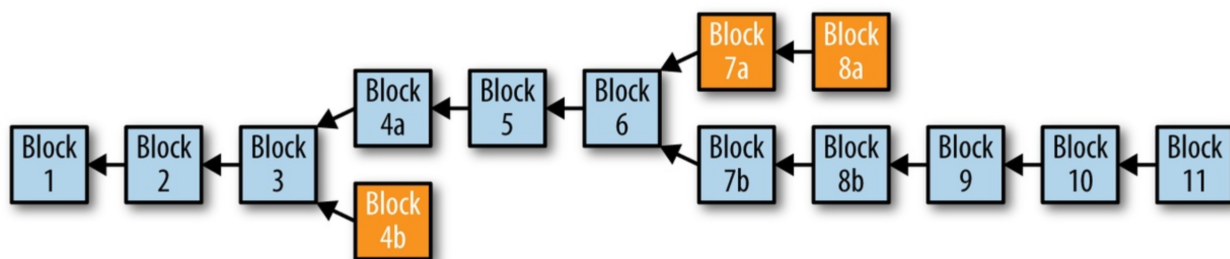
10.13.1硬分叉

在前面章节比特币分叉中，我们研究了比特币网络如何短暂地分叉，网络中的两个部分在短时间内处于区块链的两个不同分支。我们看到这个过程是如何自然发生的，作为网络的正常运行的一部分，以及如何在一个或多个块被挖掘之后，网络在一个统一的区块链上重新收敛。

另一种情况是，网络也可能会分叉到两条链条，这是由于共识规则的变化。这种分叉称为硬分叉，因为这种分叉后，网络不会重新收敛到单个链路上。相反，这两条链子独立发展。当比特币网络的一部分节点按照与网络的其余部分节点不同的一致性规则运行时，硬分叉就会发生。

这可能是由于错误或者由于故意改变了协商一致规则的实施而发生的。硬分叉可用于改变共识的规则，但需要在系统中所有参与者之间进行协调。没有升级到新的共识规则的任何节点都不能参与共识机制，并且在硬分叉时刻被强制到单独的链上。

因此，硬分叉引入的变化可以被认为不是“向前兼容”，因为未升级的系统不能再处理新的共识规则。让我们来看一下硬分叉的技巧和具体的例子。下图10-9显示区块链出现两个分叉。在块高度4处，发生单一个区块分叉。这是我们在前面章节比特币分叉中看到的自发分叉的类型。经过块5的挖掘，网络在一条链上重新收敛，分叉被解决。



然而，后来在块高度6处发生了硬分叉。我们假设原因是由于新共识规则的变化出现新的客户端版本。从块高度7开始，矿工运行新的版本，需要接受新类型的数字签名，我们称之为“Smores”签名，它不是基于ECDSA的签名。紧接着，运行新版本的节点创建了一笔包含Smores签名的交易，一个更新了软件的矿工挖矿出了包含此交易的区块7b。

任何尚未升级软件以验证Smores签名的节点或矿工现在都无法处区块7b。从他们的角度来看，包含Smores签名的交易和包含该交易的块7b的交易都是无效的，因为它们是根据旧的共识规则进行评估的。

这些节点将拒绝该笔交易和区块，并且不会传播它们。正在使用旧规则的任何矿工都不接受块7b，并且将继续挖掘其父级为块6的候选块。实际上，使用如果它们连接的所有节点都是也遵守旧的规则，遵守旧规则的矿工甚至可能接收不到块7b，因此不会传播这个区块。最终，他们将能够开采区块7a，这个旧的规则是有效的，其中不包含与Smores签名的任何交易。

这两个链条从这一点继续分歧。“b”链的矿工将继续接受并开采含有Smores签名的交易，而“a”链上的矿工将继续忽视这些交易。即使块8b不包含任何Smores签名的交易，“a”链上的矿工也无法处理。对他们来说，它似乎是一个孤立的块，因为它的父“7b”不被识别为一个有效的块。

10.13.2硬分叉：软件，网络，采矿和链

对于软件开发人员来说，术语“分叉”具有另一个含义，对“硬分叉”一词增加了混淆。在开源软件中，当一组开发人员选择遵循不同的软件路线图并启动开源项目的竞争实施时，会发生叉。我们已经讨论了两种情况，这将导致硬分叉：共识规则中的错误，以及对共识规则的故意修改。在故意改变共识规则的情况下，软件叉在硬分叉之前。但是，对于这种类型的硬分叉，新的共识规则必须通过开发，采用和启动新的软件实现。

试图改变共识规则的软分叉的例子包括Bitcoin XT，Bitcoin Classic和最近的Bitcoin Unlimited。但是，这些软分叉都没有产生硬分叉。虽然软件叉是一个必要的前提条件，但它本身不足以发生硬分叉。对于硬分叉发生，必须是由于采取相互竞争的实施方案，并且规则需要由矿工，钱包和中间节点激活。相反，有许多比特币核心的替代实现方案，甚至还有软分叉，这些没有改变共识规则，阻止发生错误，可以在网络上共存并互操作，最终并未导致硬分叉。

不同的共识规则在交易或块的验证中会以明确的和清晰的方式有所不同。这种差别可能以微妙的方式表现，比如共识规则适用于比特币脚本或加密原语（如数字签名）时。最后，共识规则的差别还可能会由于意料之外的方式，比如由于系统限制或实现细节所产生的隐含共识约束。在将Bitcoin Core 0.7升级到0.8之前的意料之中的硬分叉中看到了后者的一个例子，这是由于用于存储块的Berkley DB实现的限制引起的。

从概念上讲，我们可以将硬分叉子看成四个阶段：软分叉，网络分叉，挖矿分叉和区块链分叉。该过程开始于开发人员创建的客户端，这个客户端对共识规则进行了修改。当这种新版本的客户端部署在网络中时，一定百分比的矿工，钱包用户和中间节点可以采用并运行该版本客户端。得到的分叉将取决于新的共识规则是否适用于区块，交易或系统其他方面。如果新的共识规则与交易有关，那么当交易被挖掘成一个块时，根据新规则创建交易的钱包可能会产生出一个网络分叉，这就是一个硬分叉。如果新规则与区块有关，那么当一个块根据新规则被挖掘时，硬分叉进程将开始。

首先，是网络分叉。基于旧的共识规则的节点将拒绝根据新规则创建的任何交易和块。此外，遵循旧的共识规则的节点将暂时禁止和断开发送这些无效交易和块的任何节点。因此，网络将分为两部分：旧节点将只保留连接到旧节点，新节点只能连接到新节点。基于新规则的单个交易或块将通过网络波动，实现分区，导致出现两个网络。

一旦使用新规则的矿工开采了一个块，挖矿和区块链也将分叉。新的矿工将在新区块之上挖掘，而老矿工将根据旧的规则挖掘一个单独的链条。处于分区的网络将使得按照各自共识规则运行的矿工将不会接收彼此的块，因为它们连接到两个单独的网络。

10.13.3分离矿工和难度

随着矿工们被分裂开始开采两条不同的链条，链上的哈希算力也被分裂。两个链之间的挖矿能力可以分成任意比例。新的规则可能只有少数人跟随，或者也可能是绝大多数矿工。

我们假设，例如，80%-20%的分割，大多数挖矿能力使用新的共识规则。我们还假设分叉在改变目标（retarget）后立即出现。这两条链将从改变目标之后各自接受自己的难度。新的共识规则拥有80%的以前可用的挖矿权的承诺。从这个链的角度来看，与上一周期相比，挖矿能力突然下降了20%。区块将会平均每12分钟发现一次，这意味着可以扩大这条链的挖矿能力下降了20%。这个块发行速度将持续下去（除非有任何改变哈希功率的因素出现），直到2016块开采，这将需要大约24,192分钟（每个块需要12分钟）或16.8天。 16.8天后，基于此链中哈希算力的减少，改变目标将再次发生，并将难度调整（减少20%）每10分钟产生一个区块。

少数人认可的那条链，根据旧规则继续挖矿，现在只有20%的哈希算力，将面临更加艰巨的任务。在这条链上，平均每隔50分钟开采一次。这个难度将不会在2016个块之前进行调整，这将需要100,800分钟，或大约10周的时间。假设每个块具有固定容量，这也将导致交易容量减少5倍，因为每小时可用于记录交易的块大幅减少了。

10.13.4有争议的硬叉

这是共识软件开发的黎明。正如开源开发改变了软件的方法和产品，创造了新的方法论，新工具和新社区，共识软件开发也代表了计算机科学的新前沿。在比特币发展路线图的辩论，实验和纠结之中，我们将看到新的开发工具，实践，方法和社区的出现。硬分叉被视为有风险，因为它们迫使少数人被迫选择升级或是必须保留在少数派链条上。将整个系统分为两个竞争系统的风险被许多人认为是不可接受的风险。结果，许多开发人员不愿使用硬分叉机制来实现对共识规则的升级，除非整个网络都能达成一致。

任何没有被所有人支持的硬分叉建议也被认为是“有争议”的，不愿冒险分裂系统。硬分叉的问题在比特币开发社区也是非常有争议的，尤其是与控制区块大小限制的共识规则的任何提议相关。一些开发商反对任何形式的硬叉，认为它太冒险了。另一些人认为硬分叉机制是提升共识规则的重要工具，避免了“技术债务”，并与过去提供了一个干净的了断。

最后，有些开发商看到硬分叉作为一种应该很少使用的机制，应该经过认真的计划，并且只有在近乎一致的共识下才建议使用。我们已经看到出现了新的方法来解决硬分叉的危险。在下一节中，我们将看一下软分叉，以及BIP-34和BIP-9信号和激活共识修改的方法。

10.13.5软分叉

并非所有共识规则的变化都会导致硬分叉。只有前向不兼容的共识规则的变化才会导致分叉。如果共识规则的改变也能够让未修改的客户端仍然按照先前的规则对待交易或者区块，那么就可以在不进行分叉的情况下实现共识修改。这就是软分叉，来区分之前的硬分叉。实

际上软分叉不是分叉。软分叉是与共识规则的前向兼容并作些变化，允许未升级的客户端程序继续与新规则同时工作。

软分叉的一个不是很明显的方面就是，软分叉级只能用于增加共识规则约束，而不是扩展它们。为了向前兼容，根据新规则创建的交易和块也必须在旧规则下有效，但是反之亦然。新规则只能增加限制，否则，根据旧规则创建的交易和区块被拒绝时，还是会将触发硬分叉。

软叉可以通过多种方式实现，该术语不定义单一方法，而是一组方法，它们都有一个共同点：它们不要求所有节点升级或强制非升级节点必须脱离共识。

10.13.5.1软分叉重新定义NOP操作码

基于NOP操作码的重新解释，在比特币中实现了一些软分叉。 Bitcoin脚本有10个操作码保留供将来使用，NOP1到NOP10。 根据共识规则，这些操作码在脚本中的存在被解释为无效的运算符，这意味着它们没有任何效果。 在NOP操作码之后继续执行，就好像它不存在一样。因此，软叉可以修改NOP代码的语义给它新的含义。

例如，BIP-65（CHECKLOCKTIMEVERIFY）重新解释了NOP2操作码。 实施BIP-65的客户将NOP2解释为OP_CHECKLOCKTIMEVERIFY，并在其锁定脚本中包含该操作码的UTXO上，强制了绝对锁定的共识规则。 这种变化是一个软分叉，因为在BIP-65下有效的交易在任何没有实现（不了解）BIP-65的客户端上也是有效的。 对于旧的客户端，该脚本包含一个NOP代码，这被忽略。

10.13.5.2其他方式软分叉升级

NOP操作码的重新解释既是计划的，也是共识升级的明显机制。然而，最近，引入了另一种软分叉机制，其不依赖于NOP操作码，用于非常特定类型的共识改变。这在隔离见证掌机[segwit]中有更详细的检查。 Segwit是一个交易结构的体系结构变化，它将解锁脚本（见证）从交易内部移动到外部数据结构（将其隔离）。

Segwit最初被设想为硬分叉升级，因为它修改了一个基本的结构（交易）。在2015年11月，一位从事比特币核心工作的开发人员提出了一种机制，通过这种机制，可以将软件包引入segwit。用于此的机制是在segwit规则下创建的UTXO的锁定脚本的修改，使得未修改的客户端将任何解锁脚本视为可锁定脚本。

因此，可以引入segwit就是软分叉，而不需要每个节点必须从链上升级或拆分网络。有可能还有其他尚未被发现的机制，通过这种机制可以以向前兼容的方式进行升级，都作为软分叉。

10.13.6对软分叉的批评

基于NOP操作码的软分叉是相对无争议的。 NOP操作码被放置在比特币脚本中，明确的目标是允许无中断升级。然而，许多开发人员担心软分叉升级的其他方法会产生不可接受的折衷。

对软分叉更改的常见批评包括：

- 技术性债务

由于软叉在技术上比硬叉升级更复杂，所以引入了技术性债务，这是指由于过去的设计权衡而增加代码维护的未来成本。代码复杂性又增加了错误和安全漏洞的可能性。

- 验证放松

未经修改的客户端将交易视为有效，而不评估修改的共识规则。实际上，未经修改的客户端不会使用全面的协商一致的规则来验证，因为它们对新规则无视。这适用于基于NOP的升级，以及其他软分叉升级。

- 不可逆转升级

因为软分叉产生额外的共识约束的交易，所以它们在实践中成为不可逆转的升级。如果软分叉升级在被激活后被回退，根据新规则创建的任何交易都可能导致旧规则下的资金损失。例如，如果根据旧规则对CLTV交易进行评估，则不存在任何时间锁定约束，并且可以花费在任何时间。因此，评论家认为，由于错误而不得不被回退的失败的软分叉几乎肯定会导致资金的流失。

10.14使用区块版本发出软分叉信号

由于软分叉允许未经修改的客户在协商一致的情况下继续运作，“激活”软分叉的机制是通过向矿工发出信号准备：大多数矿工必须同意他们准备并愿意执行新的共识规则。 为了协调他们的行动，有一个信号机制，使他们能够表达他们对共识规则改变的支持。 该机制是在2013年3月激活BIP-34并在2016年7月被BIP-9激活所取代。

10.14.1 BIP-34信号和激活

在BIP-34中的第一个实施使用块版本字段来允许矿工表示准备达成特定的共识规则更改。在BIP-34之前，按照惯例将块版本设定为“1”，而不是以共识方式执行。BIP-34定义了一个共识规则变更，要求将Coinbase交易的coinbade字段（输入）包含块高度。在BIP-34之前，Coinbase可以让矿工选择包含的任意数据。在BIP-34激活之后，有效块必须在Coinbase的开始处包含特定的块高度，并且使用大于或等于“2”的版本号进行标识。

为了标记BIP-34的更改和激活，矿工们将块版本设置为“2”而不是“1”。这没有立即使版本“1”块无效。一旦激活，版本“1”块将变得无效，并且将需要所有版本“2”块以包含Coinbase库中的块高度才能有效。

BIP-34基于1000个块的滚动窗口定义了两步启动机制。矿工将以“2”作为版本号来构建块，从而向BIP-34发出信号。严格来说，由于共识规则尚未被激活，这些区块还没有遵守新的共识规则，也就是将块高度包括在Coinbase交易中。

共识规则分为两个步骤激活：

如果75%（最近1000个块中的750个）标有版本“2”，则版本“2”块必须包含coinbase交易中的块高度，否则被拒绝为无效。 版本“1”块仍然被网络接受，不需要包含块高度。 这个新时期的新旧共识规则共存。

当95%（最近1000块中的950）是版本“2”时，版本“1”块不再被视为有效。 版本“2”块只有当它们包含coinbase中的块高度（根据先前阈值）时才有效。 此后，所有块必须符合新的一致性规则，所有有效块必须包含coinbase交易中的块高度。根据BIP-34规则成功发信号和激活后，该机制再次使用两次以激活软分叉：BIP-66标签的严格DER编码通过BIP-34信号通过块版本“3”激活，无效版本“2”块。BIP-65 CHECKLOCKTIMEVERIFY被块版本“4”的BIP-34信号激活，无效版本“3”块。

BIP-65激活后，BIP-34的信号和激活机制退出，并用下面描述的BIP-9信号传导机制代替。这个标准在BIP-34(Block v2, Height in Coinbase)中定义。

10.14.2 BIP-9信号和激活

BIP-34，BIP-66和BIP-65使用的机制成功地激活了三个软分叉。 然而，它又被替换了，因为它有几个限制：

通过使用块版本的整数值，一次只能激活一个软分叉，因此需要软分叉提议之间的协调以及对其优先级和排序的协议。

此外，由于块版本增加，所以机制并没有提供一种直接的方式来拒绝变更，而是提出一个不同的方法。 如果老客户仍然在运行，他们可能会错误地将信号转换为新的更改，作为以前拒绝的更改的信号。

每个新的更改不可逆转地减少可用的供将来更改的块版本。

提出BIP-9来克服这些挑战，提高实施未来变化的速度和便利性。

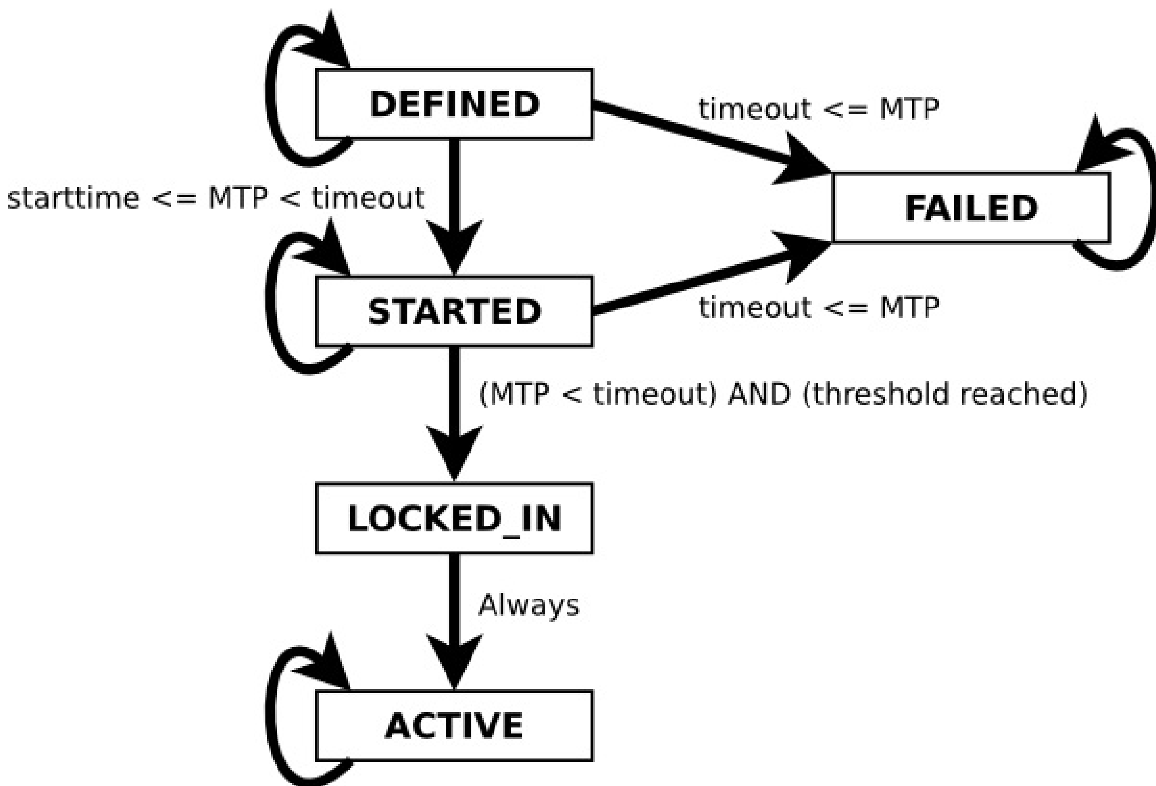
BIP-9将块版本解释为bit字段而不是整形（int）字段。因为块版本最初作为整形（int32）字段，因此版本1到4，只有29位可用作bit字段。这留下29位，可以独立使用，同时在29个不同的提案上表示准备就绪。BIP-9还设置了信令和激活的最大时间。矿工们不需要永远发出信号。如果提案在TIMEOUT期间（在提案中定义）未激活，则该提案被视为被拒绝。该提议可以使用不同位的信令重新提交，更新激活周期。

此外，在TIMEOUT已经过去并且特征被激活或被拒绝之后，信令位可以被再次用于另一个特征而不会混淆。因此，多达29次更改可以并行发出信号，TIMEOUT后可将这些位“再循环”以提出新的更改。

注意虽然信令位可以重复使用或回收利用，但只要投票期间不重叠，BIP-9的作者就建议仅在必要时重复使用位; 主要是由于旧软件中的bug，可能会发生意外的行为。 总之，我们不应该期望在所有29位都被使用一次之前看到重用。

建议的更改由包含以下字段的数据结构标识：名称用于区分提案的简短描述。 通常，BIP将该提案描述为“bipN”，其中N是BIP编号。位0到28，矿工使用的块版本中的位用于表示此提案的批准。开始时间信号开始的时间（基于中值时间过去或MTP），之后该位的值被解释为提示的信令准备。时间结束该时间（基于MTP），如果尚未达到激活阈值，则认为该更改被拒绝。

与BIP-34不同，BIP-9根据2016块的难度改变目标（retarget）周期，在整个间隔中计数激活信号。 对于每个改变目标期间，如果提案的信号块的总和超过95%（2016中的1916），则该提案将在稍后的改变目标期间激活。BIP-9提供了一个提案状态图，以说明一个提案的各个阶段和转换，如图10-10所示。



一旦它们的参数在比特币软件中被知道（定义），提案将在DEFINED状态下开始。 对于具有MTP的块在开始时间之后，提议状态转换到STARTED。 如果在改变目标期间超过了投票阈值，并且未超过超时，则提案状态转换为LOCKED_IN。 一个改变目标期后，该提案变为活动。

一旦达到这个状态，提案仍然处于活动状态。 如果在达到投票阈值之前超时时间，提案状态将更改为“已故”，表示已拒绝的提案。 REJECTED的建议永远在这个状态。BIP-9首次实施用于激活CHECKSEQUENCEVERIFY和相关BIP（68,112,113）。 名为“csv”的建议在2016年7月成功启动。标准定义在BIP-9(Version bits with timeout and delay).

10.15共识软件开发

共识软件开发不断发展，对于改变共识规则的各种机制进行了很多讨论。由于其本质，比特币在协调和变化共识方面树立了非常高的标杆。作为一个去中心化的制度，不存在将权力强加于网络参与者的“权威”。权力分散在多个支持者，如矿工，核心开发商，钱包开发商，交易所，商家和最终用户之间。这些支持者不能单方面做出决定。例如，矿工在理论上可以通过简单多数（51%）来改变规则，但受到其他支持者的同意的限制。

如果他们单方面行事，其他参与者可能会拒绝遵守，将经济活动保持在少数链条上。没有经济活动（交易，商人，钱包，交易所），矿工们将用空的块开采一个无价值的货币。这种权力的扩散意味着所有参与者必须协调，或者不能做出任何改变。现状是这个制度的稳定状态，只有在很大程度上达成一致的情况下，才能有几个变化。软分叉的95%门槛反映了这一现实。

重要的是要认识到，对于共识发展没有完美的解决办法。硬分叉和软分叉都涉及权衡。对于某些类型的更改，软分叉可能是一个更好的选择;对于别人来说，硬分叉可能是一个更好的选择。没有完美的选择，都带有风险。共识软件开发的一个不变特征是变革是困难的，是共识力量的妥协。有些人认为这是共识制度的弱点。在时间上，你可以像我一样看到它，把它当作这个系统最大的优势。

保全比特币是很有挑战性的事，因为比特币不像银行账户余额那样体现抽象价值。比特币其实更像数字现金或黄金。 你可能听过这样的说法，“谁持有几乎等同法律拥有。”好吧，在比特币的世界里，谁持有谁拥有。持有拥有解锁比特币的密钥就相当于持有现金或一块贵重金属。你可能会将密钥丢失，会放错地方，会被盗或者不小心错支了数额。无论是哪种场景，用户都没有办法撤回，因为这就像是将现金丢在了车水马龙的大街上。

不过，与现金、黄金或者银行账户相比，比特币有着一个独一无二的优势。你不能“备份”你的现金、黄金或者银行账 户，但你可以像备份其他文件一样，备份含有密钥的比特币钱包。它可以被复制成很多份，放到不同的地方保存起来， 甚至能打印到纸上进行实体备份。比特币与至今为止的其他货币是如此不同，以致于我们需要以一种全新的思维方式来 衡量比特币的安全性。

11.1 安全准则

比特币的核心准则是去中心化，这一点对安全性具有重要意义。在中心化的模式下，例如传统的银行或支付网络，需要依赖于访问控制和审查制度将不良行为者拒之门外。相比之下，比特币这样的去中心化系统则将责任和控制权都移交给了用户。由于网络的安全性是基于工作量证明而非访问控制，比特币网络可以对所有人开放，也无需对比特币传输进行加密。

在一个传统的支付网络中，例如信用卡系统，支付是终端开放式的，因为它包含了用户的个人标识（信用卡号）。在初次支付后，任何能获得该标识的人都可以从所有者那里反复“提取”资金。因此，该支付网络必须采取端对端加密的方式，以确保没有窃听者或中间人可以在资金流通或存储过程中将交易数据截获。如果坏人获得该系统的控制权，他将能破获当前的的交易和支付令牌，他还可以随意动用这笔资金。更糟的是，当客户数据被泄露时，顾客的个人身份信息将被盗窃者们一览无余。客户这时必须立即采取措施，以防失窃帐户被盗窃者用于欺诈。

比特币则截然不同，一笔比特币交易只授权向指定接收方发送一个指定数额，并且不能被修改或伪造。它不会透露任何个人信息，例如当事人的身份，也不能用于权限外的支付。因此，比特币的支付网络并不需要加密或防窃听保护。事实上，你可以在任何公开的网络上广播比特币交易的数据，例如在不安全的WIFI或蓝牙网络上公开传播比特币交易的数据，这对安全性没有任何影响。

比特币的去中心化安全模型很大程度上将权力移交到用户手上，随之而来的是用户们保管好密钥的责任。这对于大多数 用户来说并非一件易事，特别是在像智能手机或笔记本电脑这种能能时刻联网的通用设备上。虽然比特币的去中心化模 型避免了常见的信用卡盗用等情况，但很多用户由于无法保管好密钥从而被黑客攻击。

11.1.1 比特币系统安全开发

对于比特币开发者而言最重要的是去中心化原则。大多数开发者对中心化的安全模型很熟悉，并可能试图将中心化的模型运用到借鉴比特币的应用中去，这将给比特币带来灭顶之灾。

比特币的安全性依赖于密钥的分散性控制，并且需要矿工们各自独立地进行交易验证。如果你想利用好比特币的安全性，你需要确保自己处于比特币的安全模型里。简而言之，不要将用户的密钥控制权拿走，不要接受非区块链交易信息。

例如，许多早期的比特币交易所将所有用户的资金集中在一个包含着私钥的“热钱包”里，并存放在服务器上。这样的设计夺取了用户的掌控权，并将密钥集中到单个系统里。很多这样的系统都被黑客攻破了，并给客户带来灾难性后果。 另一个常见的错误是接受区块链离线交易，妄图减少交易费或加速交易处理速度。一个“区块链离线交易”系统将交易数据记录在一个

内部的中心化账本上，然后偶尔将它们同步到比特币区块链中。这种做法，再一次，用专制和集中的方式取 代比特币的去中心化安全模型。当数据处于离线的区块链上的时候，保护不当的中心化账本里的资金可能会不知不觉被 伪造、被挪用、被消耗。

除非你是准备大力投资运营安全，叠加多层访问控制，或（像传统的银行那样）加强审计，否则在将资金从比特币的去中心化安全场景中抽离出来之前，你应该慎重考虑一番。即使你有足够的资金和纪律去实现一个可靠的安全模型，这样的设计也仅仅是复制了一个脆弱不堪，深受账户盗窃威胁、贪污和挪用公款困扰的传统金融网络而已。要想充分利用比特币特有的去中心化安全模型，你必须避免中心化架构的常见诱惑，因它最终将摧毁比特币的安全性。

11.1.2 信任根

传统的安全体系基于一个称为信任根（ROOT OF TRUST）的概念，它指的总体系统或应用程序中一个可信赖的安全核心。安全体系像一圈同心圆一样围绕着信任根源来进行开发，像层层包裹的洋葱一样，信任从内至外依次延伸。每一层都构建于更可信的内层之上，通过访问控制，数字签名，加密和其他安全方式确保可信。随着软件系统变得越来越复杂，它们更可能出现问题，安全更容易受到威胁。其结果是，软件系统变得越复杂，就越难维护安全性。信任根的概念确保绝大多数的信任被置于一个不是过于复杂系统的一部分，因此该系统的这部分也相对坚固，而更复杂的软件则在它之上构建。这样的安全体系随着规模扩大而不断重复出现，首先信任根建立于单个系统的硬件内，然后将该信任根通过操作系统扩展到更高级别的系统服务，最后逐次扩散到圈内多台服务器上。

比特币的安全体系与这不同。在比特币里，共识系统创建了一个可信的完全去中心化的公开账本，一个正确验证过的区块使用创世区块 作为信任根，建立一条至当前区块的可信任链。比特币系统可以使用区块链作为它们的信任根。在设计一个多系统服务机制的比特币应用时，你应该仔细确认安全体系，以确保对它的信任能有据可依。最终， 唯一可确信无疑的是一条完全有效的区块链。如果你的应用程序或明或暗地信赖于区块链以外的东西，就该引起重视， 因为它可能会引入漏洞。一个不错的方法评估你应用程序的安全体系：单独考量每个组件，设想该组件被完全攻破并被坏人掌控的场景。依次取出应用程序的每个组件，并评估它被攻破时对整体安全的影响。如果你的应用程序的安全性在该组件沦陷后大打折扣，那就说明你已经对这些组件过度信任了。一个没有漏洞的比特币应用程序应该只受限于比特币的共识机制，这意味着其安全体系的信任源于比特币最底层的部分。

无数个黑客攻击比特币交易所的例子都是因为轻视了这一点，他们的安全体系和设计甚至无法通过基本的审查。这种中心化的实现方式将信任置于比特币区块链之外的诸多组件之上，例如热钱包，中心化的账本数据库，简易加密的密钥，以及许多类似的方案。

11.2 用户最佳安全实践

人类使用物理的安全控制已经有数千年之久。相比之下，我们的数字化安全经验的年纪还不满50岁。现代通用的操作系统并不是十分安全，亦不特别适合用来存储数字货币。我们的电脑通过一直连接的互联网长时间暴露在外，它们运行着成千上万第三方软件组件，这些软件往往可以不受约束地访问用户的文件。你电脑上安装的众多软件只要有一个恶意软件，就会威胁到你的文件，可窃取你钱包里的所有比特币。想要杜绝病毒和木马对电脑的威胁，用户要达到一定的计算机维护水平，只有小部分人能做到。

尽管信息安全经过了数十年的研究和发展，数字资产在绵延不绝的攻势下还是十分脆弱。纵使是像金融服务公司，情报机构或国防承包商这样拥有高度防护和限制的系统，也经常会被攻破。比特币创造了具有内在价值的数字资产，它可以被窃取，并立即转移给他人而无法撤回。这让黑客有了强烈的作案动机。至今为止，黑客都不得不在套现后更换身份信 息或帐户口令，例如信用卡或银行账户。尽管掩饰和洗白这部分财务信息的难度不小，但越来越多的窃贼从于此道。而 比特币使这个问题加剧了，因为它不需要掩饰或洗白，它本身就是具有内在价值的数字资产。

幸运的是，比特币也有着激励机制，以提高计算机的安全性。如前所述，计算机受威胁的风险是模糊的，间接的，而比特币让这些风险变得明确清晰。在电脑上保存比特币让用户时刻注意他们需要提高计算机的安全性，结果便是这使得比特币和其它数字货币得以传播和扩散，我们已经看到在黑客技术和安全解决方案双方的提升。简单来说，黑客现在有着一个非常诱人的目标，而用户也有明确的激励性去保卫自己。

在过去的三年里，随着比特币不断被接纳，一个直接的结果是，我们已经看到信息安全领域取得了巨大创新，例如硬件加密，密钥存储和硬件钱包，多重签名技术和数字托管。在下面的章节中，我们将研究各种实际用户安全中的实践经验。

11.2.1 比特币物理存储

相比数字信息的安全，大多数用户对物理安全更加熟悉，一个非常有效保护比特币的方法是，将它们转换为物理形式。 比特币密钥不过是串长数字而已。这意味着它们可以以物理形式存储起来，如印在纸上或蚀刻成金属硬币上。这样保护密钥就变成了简单地保护印着比特币密钥的物理实体。一组打印在纸上的比特币密钥被称为“纸钱包”，有许多可以用来创建它们的免费工具。我个人将大部分（99%以上）的比特币存储在纸钱包上，并用BIP0038加密，复制了多份并锁在保险箱里。将比特币离线保存的方法被称为冷存储，它是最有效的安全技术之一。冷存储系统是在一个离线系统（一个从来没有连接过互联网的系统）上生成密钥，并离线存储到纸上或者U盘等电子媒介上。

11.2.2 硬件钱包

从长远来看，比特币安全将越来越多地以硬件防篡改钱包的形式出现。与智能手机或台式电脑不同，一个比特币硬件钱包只有一个目的，安全地存储比特币。不像容易受害的常用软件那样，硬件钱包只提供了有限的接口，从而可以给非专业用户提供近乎万无一失的安全等

级。我预期将看到硬件钱包成为比特币储存的主要方式。要想看硬件钱包的实例，请查阅TREZOR。

11.2.3 平衡风险

虽然大多数用户都非常关注比特币防盗，其实还有一个更大的风险存在。数据文件丢失的情况时有发生。如果比特币的数据也在其中，损失将会让人痛苦不堪。为了保护好比特币钱包，用户必须非常注意不要剑走偏锋，这样不至于会搞丢比特币。在2011年7月，一个著名的比特币认知教育项目损失了近7,000枚比特币。为了防止被盗窃，其主人曾之前采取了一系列复杂的操作去加密备份。结果他们不慎丢失了加密的密钥，使得备份变得毫无价值，白白失去了一大笔财富。如果你保护比特币的方式太过了，这好比于把钱藏在沙漠里，你可能不能再把它找回来了。

11.2.4 分散风险

你会将你的全部家当换成现金放在钱包里随身携带么？大多数人会认为这非常不明智，但比特币用户经常会将所有的比特币放在一个钱包里。用户应该将风险分散到不同类型的比特币钱包。审慎的用户应该只留一小部分（或许低于5%）的比特币在一个在线的或手机钱包，就像零用钱一样，其余的部分应该采用不同存储机制分散开来，诸如电脑钱包和离线（冷存储）钱包。

11.2.5 多重签名管理

当一个公司或个人持有大量比特币时，他们应该考虑采用多重签名的比特币地址。多重签名比特币地址需要多个签名才能支付，从而保证资金的安全。多重签名的密钥应存储在多个不同的地方，并由不同的人掌控。打个比方，在企业环境中，密钥应该分别生成并由若干公司管理人员持有，以确保没有任何一个人可以独自占有资金。多重签名的地址也可以提供冗余，例如一个人持有多个密钥，并将它们分别存储在不同的地方。

11.2.6 存活能力

一个非常重要却又常常被忽视的安全性考虑是可用性，尤其是在密钥持有者丧失工作能力或死亡的情况下。比特币的用户被告知应该使用复杂的密码，并保证他们的密钥安全且不为他人所知。不幸的是，这种做法使得在用户无法解锁时， 用户的家人几乎无法将该财产恢复。事实上，比特币用户的家人可能完全不知道这笔比特币资金的存在。 如果你有很多的比特币，你应该考虑与一个值得信赖的亲属或律师分享解密的细节。可以搞一个更复杂的比特币恢复计划，可以通过设置多重签名，做好遗产规划，并通过专门的“数字资产执行者”律师处理后事。

11.3 总结

比特币是一项全新的，前所未有的，复杂的技术。随着时间的推移，我们将开发出更好的安全工具，而且更容易被非专业人士使用。而现在，比特币用户可以使用许多这里所讨论的技巧，享受安全而无困扰的比特币生活。

附录1、比特币白皮书：比特币白皮书：一种点对点的电子现金系统

原文作者：中本聪（**Satoshi Nakamoto**）

作者邮箱：**Satoshin@gmx.com**

执行翻译：**8btc.com** 巴比特 **QQagent**

> [摘要]：本文提出了一种完全通过点对点技术实现的电子现金系统，它使得在线支付能够直接由一方发起并支付给另外一方，中间不需要通过任何的金融机构。虽然数字签名（Digital signatures）部分解决了这个问题，但是如果仍然需要第三方的支持才能防止双重支付（double-spending）的话，那么这种系统也就失去了存在的价值。我们(we)在此提出一种解决方案，使现金系统在点对点的环境下运行，并防止双重支付问题。该网络通过随机散列（hashing）对全部交易加上时间戳（timestamps），将它们合并入一个不断延伸的基于随机散列的工作量证明（proof-of-work）的链条作为交易记录，除非重新完成全部的工作量证明，形成的交易记录将不可更改。最长的链条不仅将作为被观察到的事件序列（sequence）的证明，而且被看做是来自CPU计算能力最大的池（pool）。只要大多数的CPU计算能力都没有打算合作起来对全网进行攻击，那么诚实的节点将会生成最长的、超过攻击者的链条。这个系统本身需要的基础设施非常少。信息尽最大努力在全网传播即可，节点(nodes)可以随时离开和重新加入网络，并将最长的工作量证明链条作为在该节点离线期间发生的交易的证明。

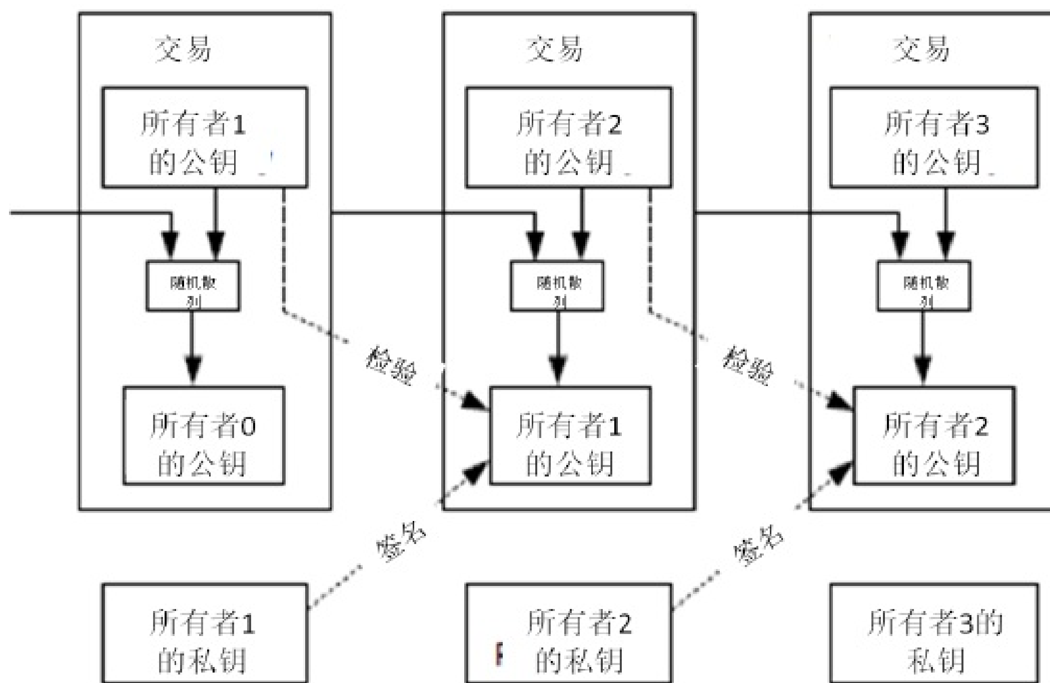
1. 简介

互联网上的贸易，几乎都需要借助金融机构作为可资信赖的第三方来处理电子支付信息。虽然这类系统在绝大多数情况下都运作良好，但是这类系统仍然内生性地受制于“基于信用的模式”(trust based model)的弱点。我们无法实现完全不可逆的交易，因为金融机构总是不可避免地会出面协调争端。而金融中介的存在，也会增加交易的成本，并且限制了实际可行的最小交易规模，也限制了日常的小额支付交易。并且潜在的损失还在于，很多商品和服务本身是无法退货的，如果缺乏不可逆的支付手段，互联网的贸易就大大受限。因为有潜在的退款的可能，就需要交易双方拥有信任。而商家也必须提防自己的客户，因此会向客户索取完全不必要的个人信息。而实际的商业行为中，一定比例的欺诈性客户也被认为是不可避免的，相关损失视作销售费用处理。而在使用物理现金的情况下，这些销售费用和支付问题上的不确定性却是可以避免的，因为此时没有第三方信用中介的存在。 所以，我们非常需要这样一种电子支付系统，它基于密码学原理而不基于信用，使得任何达成一致的双方，能够直接进行支付，从而不需要第三方中介的参与。杜绝回滚(reverse)支付交易的可能，这就可以保护特定的卖家免于欺诈；而对于想要保护买家的人来说，在此环境下设立通常的第三方担保机

制也可谓轻松加愉快。在这篇论文中，我们(we)将提出一种通过点对点分布式的时间戳服务器来生成依照时间前后排列并加以记录的电子交易证明，从而解决双重支付问题。只要诚实的节点所控制的计算能力的总和，大于有合作关系的(cooperating)攻击者的计算能力的总和，该系统就是安全的。

2. 交易(Transactions)

我们定义，一枚电子货币（an electronic coin）是这样的一串数字签名：每一位所有者通过对前一次交易和下一位拥有者的公钥(Public key) 签署一个随机散列的数字签名，并将这个签名附加在这枚电子货币的末尾，电子货币就发送给了下一位所有者。而收款人通过对签名进行检验，就能够验证该链条的所有者。

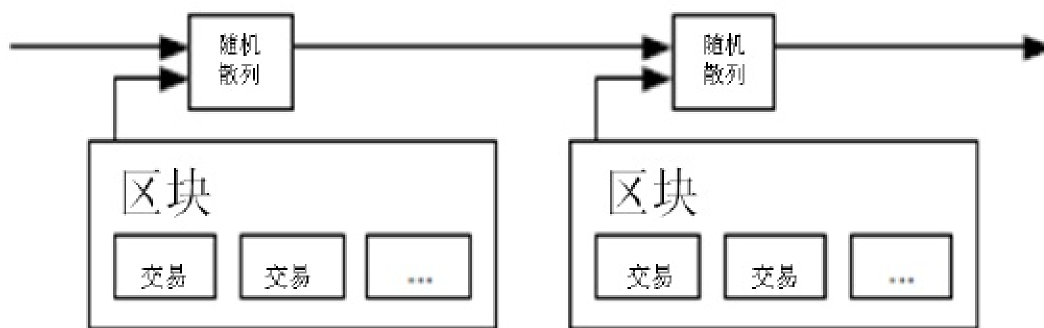


该过程的问题在于，收款人将难以检验，之前的某位所有者，是否对这枚电子货币进行了双重支付。通常的解决方案，就是引入信得过的第三方权威，或者类似于造币厂(mint)的机构，来对每一笔交易进行检验，以防止双重支付。在每一笔交易结束后，这枚电子货币就要被造币厂回收，而造币厂将发行一枚新的电子货币；而只有造币厂直接发行的电子货币，才算作有效，这样就能够防止双重支付。可是该解决方案的问题在于，整个货币系统的命运完全依赖于运作造币厂的公司，因为每一笔交易都要经过该造币厂的确认，而该造币厂就好比是一家银行。 我们需要收款人有某种方法，能够确保之前的所有者没有对更早发生的交易实施签名。从逻辑上看，为了达到目的，实际上我们需要关注的只是于本交易之前发生的交易，而不需要关注这笔交易发生之后是否会有双重支付的尝试。为了确保某一次交易是不存在的，那么唯一的方法就是获悉之前发生过的所有交易。在造币厂模型里面，造币厂获悉所有的交易，并且决定了交易完成的先后顺序。如果想要在电子系统中排除第三方中介机构，那么交

易信息就应当被公开宣布（publicly announced）[1]，我们需要整个系统内的所有参与者，都有唯一公认的历史交易序列。收款人需要确保在交易期间绝大多数的节点都认同该交易是首次出现。

3. 时间戳服务器(Timestamp server)

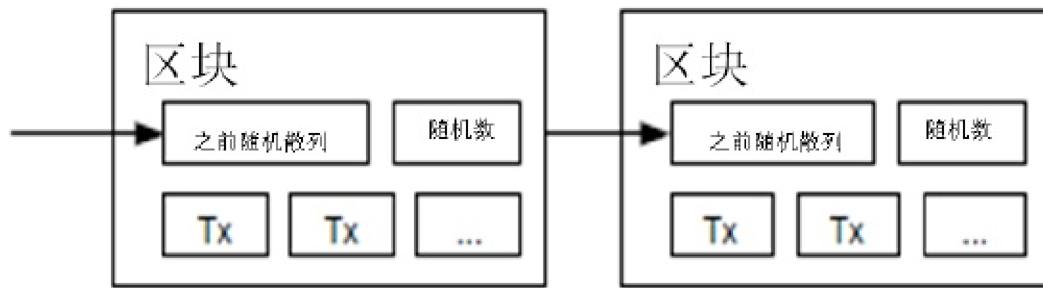
本解决方案首先提出一个“时间戳服务器”。时间戳服务器通过对以区块(block)形式存在的一组数据实施随机散列而加上时间戳，并将该随机散列进行广播，就像在新闻或世界性新闻组网络（Usenet）的发帖一样[2][3][4][5]。显然，该时间戳能够证实特定数据必然于某特定时间是的确存在的，因为只有在该时刻存在了才能获取相应的随机散列值。每个时间戳应当将前一个时间戳纳入其随机散列值中，每一个随后的时间戳都对之前的一个时间戳进行增强(reinforcing)，这样就形成了一个链条（Chain）。



4. 工作量证明（Proof-of-Work）

为了在点对点的基础上构建一组分散化的时间戳服务器，仅仅像报纸或世界性新闻网络组一样工作是不够的，我们还需要一个类似于亚当•柏克（Adam Back）提出的哈希现金（Hashcash）[6]。在进行随机散列运算时，工作量证明机制引入了对某一个特定值的扫描工作，比方说SHA-256下，随机散列值以一个或多个0开始。那么随着0的数目的上升, 找到这个解所需要的工作量将呈指数增长，而对结果进行检验则仅需要一次随机散列运算。

我们在区块中补增一个随机数(Nonce)，这个随机数要使得该给定区块的随机散列值出现了所需的那么多个0。我们通过反复尝试来找到这个随机数，直到找到为止，这样我们就构建了一个工作量证明机制。只要该CPU耗费的工作量能够满足该工作量证明机制，那么除非重新完成相当的工作量，该区块的信息就不可更改。由于之后的区块是链接在该区块之后的，所以想要更改该区块中的信息，就还需要重新完成之后所有区块的全部工作量。



同时，该工作量证明机制还解决了在集体投票表决时，谁是大多数的问题。如果决定大多数的方式是基于IP地址的，一IP地址一票，那么如果有人拥有分配大量IP地址的权力，则该机制就被破坏了。而工作量证明机制的本质则是一CPU一票。“大多数”的决定表达为最长的链，因为最长的链包含了最大的工作量。如果大多数的CPU为诚实的节点控制，那么诚实的链条将以最快的速度延长，并超越其他的竞争链条。如果想要对业已出现的区块进行修改，攻击者必须重新完成该区块的工作量外加该区块之后所有区块的工作量，并最终赶上和超越诚实节点的工作量。我们将在后文证明，设想一个较慢的攻击者试图赶上随后的区块，那么其成功概率将呈指数化递减。 另一个问题是，硬件的运算速度在高速增长，而节点参与网络的程度则会有所起伏。为了解决这个问题，工作量证明的难度(the proof-of-work difficulty)将采用移动平均目标的方法来确定，即令难度指向令每小时生成区块的速度为某一个预定的平均数。如果区块生成的速度过快，那么难度就会提高。

5. 网络

运行该网络的步骤如下：

- 1) 新的交易向全网进行广播；
- 2) 每一个节点都将收到的交易信息纳入一个区块中；
- 3) 每个节点都尝试在自己的区块中找到一个具有足够难度的工作量证明；
- 4) 当一个节点找到了一个工作量证明，它就向全网进行广播；
- 5) 当且仅当包含在该区块中的所有交易都是有效的且之前未存在过的，其他节点才认同该区块的有效性；
- 6) 其他节点表示他们接受该区块，而表示接受的方法，则是在跟随该区块的末尾，制造新的区块以延长该链条，而将被接受区块的随机散列值视为先于新区快的随机散列值。

节点始终都将最长的链条视为正确的链条，并持续工作和延长它。如果有两个节点同时广播不同版本的新区块，那么其他节点在接收到该区块的时间上将存在先后差别。当此情形，他们将在率先收到的区块基础上进行工作，但也会保留另外一个链条，以防后者变成最长的链条。该僵局（tie）的打破要等到下一个工作量证明被发现，而其中的一条链条被证实为是较长的一条，那么在另一条分支链条上工作的节点将转换阵营，开始在较长的链条上工作。 所谓“新的交易要广播”，实际上不需要抵达全部的节点。只要交易信息能够抵达足够多的节点，

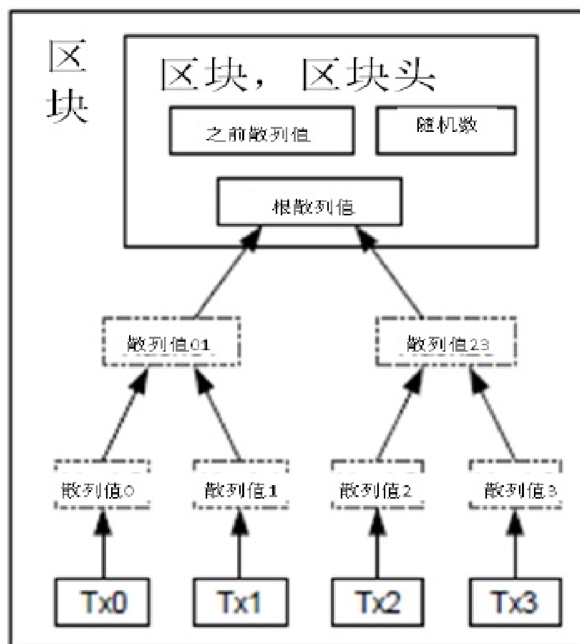
那么他们将很快被整合进一个区块中。而区块的广播对被丢弃的信息是具有容错能力的。如果一个节点没有收到某特定区块，那么该节点将会发现自己缺失了某个区块，也就可以提出自己下载该区块的请求。

6. 激励

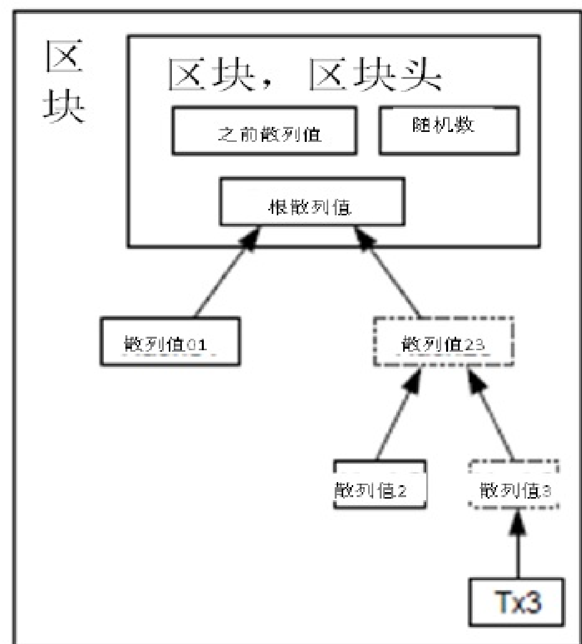
我们约定如此：每个区块的第一笔交易进行特殊化处理，该交易产生一枚由该区块创造者拥有的新的电子货币。这样就增加了节点支持该网络的激励，并在没有中央集权机构发行货币的情况下，提供了一种将电子货币分配到流通领域的一种方法。这种将一定数量新货币持续增添到货币系统中的方法，非常类似于耗费资源去挖掘金矿并将黄金注入到流通领域。此时，CPU的时间和电力消耗就是消耗的资源。 另外一个激励的来源则是交易费（transaction fees）。如果某笔交易的输出值小于输入值，那么差额就是交易费，该交易费将被增加到该区块的激励中。只要既定数量的电子货币已经进入流通，那么激励机制就可以逐渐转换为完全依靠交易费，那么本货币系统就能够免于通货膨胀。 激励系统也有助于鼓励节点保持诚实。如果有一个贪婪的攻击者能够调集比所有诚实节点加起来还要多的CPU计算力，那么他就面临一个选择：要么将其用于诚实工作产生新的电子货币，或者将其用于进行二次支付攻击。那么他就会发现，按照规则行事、诚实工作是更有利可图的。因为该等规则使得他能够拥有更多的电子货币，而不是破坏这个系统使得其自身财富的有效性受损。

7. 回收硬盘空间

如果最近的交易已经被纳入了足够多的区块之中，那么就可以丢弃该交易之前的数据，以回收硬盘空间。为了同时确保不损害区块的随机散列值，交易信息被随机散列时，被构建成一种Merkle树（Merkle tree）[7] 的形态，使得只有根(root)被纳入了区块的随机散列值。通过将该树（tree）的分支拔除（stubbing）的方法，老区块就能被压缩。而内部的随机散列值是不必保存的。



以Merkle树形式散列的交易

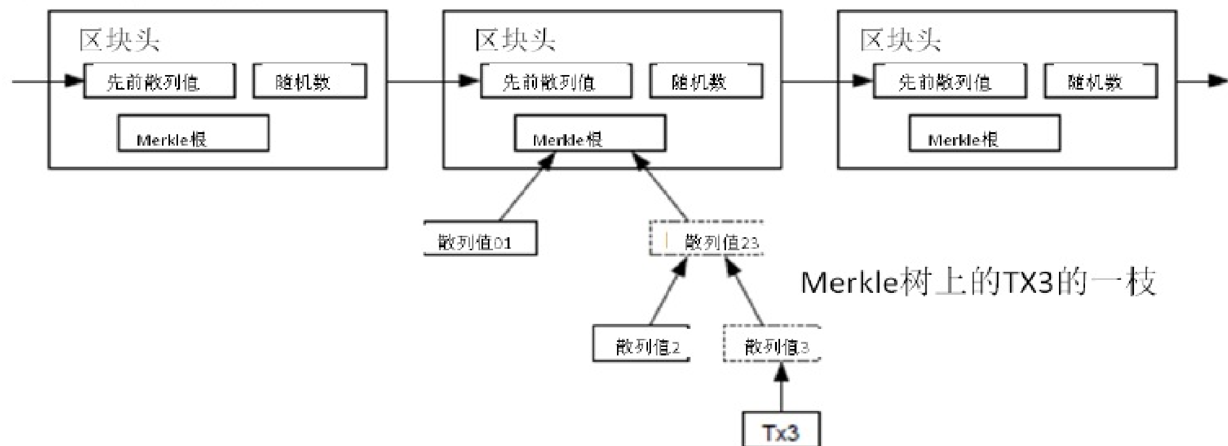


将Tx0-2从区块中剪除

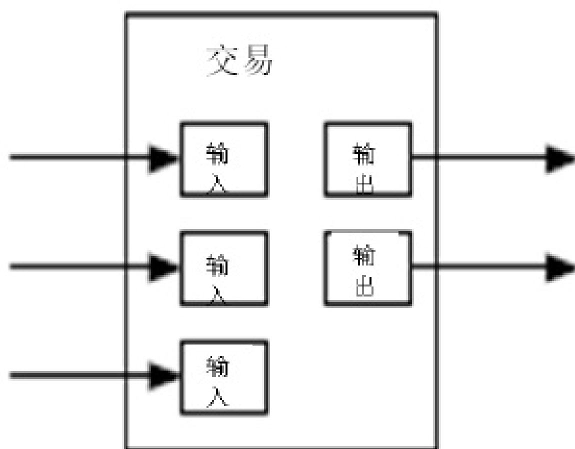
不含交易信息的区块头（Block header）大小仅有80字节。如果我们设定区块生成的速率为每10分钟一个，那么每一年产生的数据位4.2MB。（80 bytes *6* 24 * 365 = 4.2MB）。2008年，PC系统通常的内存容量为2GB，按照摩尔定律的预言，即使将全部的区块头存储于内存之中都不是问题。

8. 简化的支付确认（**Simplified Payment Verification**）

在不运行完整网络节点的情况下，也能够对支付进行检验。一个用户需要保留最长的工作量证明链条的区块头的拷贝，它可以不断向网络发起询问，直到它确信自己拥有最长的链条，并能够通过merkle的分支通向它被加上时间戳并纳入区块的那次交易。节点想要自行检验该交易的有效性原本是不可能的，但通过追溯到链条的某个位置，它就能看到某个节点曾经接受过它，并且于其后追加的区块也进一步证明全网曾经接受了它。



当此情形，只要诚实的节点控制了网络，检验机制就是可靠的。但是，当全网被一个计算力占优的攻击者攻击时，将变得较为脆弱。因为网络节点能够自行确认交易的有效性，只要攻击者能够持续地保持计算力优势，简化的机制会被攻击者焊接的（fabricated）交易欺骗。那么一个可行的策略就是，只要他们发现了一个无效的区块，就立刻发出警报，收到警报的用户将立刻开始下载被警告有问题的区块或交易的完整信息，以便对信息的不一致进行判定。对于日常会发生大量收付的商业机构，可能仍会希望运行他们自己的完整节点，以保持较大的独立完全性和检验的快速性。



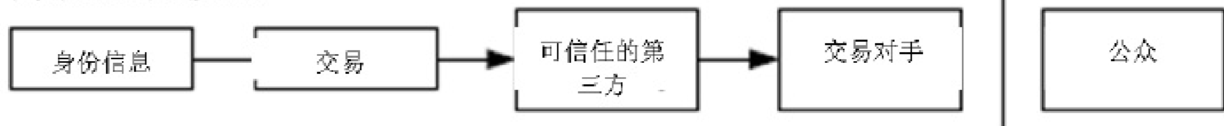
9. 价值的组合与分割（Combining and Splitting Value）

虽然可以单个单个地对电子货币进行处理，但是对于每一枚电子货币单独发起一次交易将是一种笨拙的办法。为了使得价值易于组合与分割，交易被设计为可以纳入多个输入和输出。一般而言是某次价值较大的前次交易构成的单一输入，或者由某几个价值较小的前次交易共

同构成的并行输入，但是输出最多只有两个：一个用于支付，另一个用于找零（如有）。需要指出的是，当一笔交易依赖于之前的多笔交易时，这些交易又各自依赖于多笔交易，但这并不存在任何问题。因为这个工作机制并不需要展开检验之前发生的所有交易历史。

10. 隐私（Privacy）

传统隐私模型



新隐私模型



传统的造币厂模型为交易的参与者提供了一定程度的隐私保护，因为试图向可信任的第三方索取交易信息是严格受限的。但是如果将交易信息向全网进行广播，就意味着这样的方法失效了。但是隐私依然可以得到保护：将公钥保持为匿名。公众得知的信息仅仅是有某个人将一定数量的货币发所给了另外一个人，但是难以将该交易同特定的人联系在一起，也就是说，公众难以确信，这些人究竟是谁。这同股票交易所发布的信息是类似的，股票交易发生的时间、交易量是记录在案且可供查询的，但是交易双方的身份信息却不予透露。作为额外的预防措施，使用者可以让每次交易都生成一个新的地址，以确保这些交易不被追溯到一个共同的所有者。但是由于并行输入的存在，一定程度上的追溯还是不可避免的，因为并行输入表明这些货币都属于同一个所有者。此时的风险在于，如果某个人的某一个公钥被确认属于他，那么就可以追溯出此人的其它很多交易。

11. 计算

设想如下场景：一个攻击者试图比诚实节点产生链条更快地制造替代性区块链。即便它达到了这一目的，但是整个系统也并非就此完全受制于攻击者的独断意志了，比方说凭空创造价值，或者掠夺本不属于攻击者的货币。这是因为节点将不会接受无效的交易，而诚实的节点永远不会接受一个包含了无效信息的区块。一个攻击者能做的，最多是更改他自己的交易信息，并试图拿回他刚刚付给别人的钱。诚实链条和攻击者链条之间的竞赛，可以用二叉树随机漫步（Binomial Random Walk)来描述。成功事件定义为诚实链条延长了一个区块，使其领先性+1，而失败事件则是攻击者的链条被延长了一个区块，使得差距-1。攻击者成功填补某一既定差距的可能性，可以近似地看做赌徒破产问题（Gambler's Ruin problem）。假定一个赌徒拥有无限的透支信用，然后开始进行潜在次数为无穷的赌博，试图填补上自己的亏空。那么我们可以计算他填补上亏空的概率，也就是该攻击者赶上诚实链条，如下所示[8]：

$p =$ 诚实节点制造出下一个节点的概率

$q =$ 攻击者制造出下一个节点的概率

$q_z =$ 攻击者最终消弭了z个区块的落后差距

$$q_z = \begin{cases} 1 & \text{if } p \le q \\ (\frac{q}{p})^z & \text{if } p > q \end{cases}$$

假定p>q，那么攻击成功的概率就因为区块数的增长而呈现指数化下降。由于概率是攻击者的敌人，如果他不能幸运且快速地获得成功，那么他获得成功的机会随着时间的流逝就变得愈发渺茫。那么我们考虑一个收款人需要等待多长时间，才能足够确信付款人已经难以更改交易了。我们假设付款人是一个支付攻击者，希望让收款人在一段时间内相信他已经付过款了，然后立即将支付的款项重新支付给自己。虽然收款人届时会发现这一点，但为时已晚。收款人生成了新的一对密钥组合，然后只预留一个较短的时间将公钥发送给付款人。这将可以防止以下情况：付款人预先准备好一个区块链然后持续地对此区块进行运算，直到运气让他的区块链超越了诚实链条，方才立即执行支付。当此情形，只要交易一旦发出，攻击者就开始秘密地准备一条包含了该交易替代版本的平行链条。 然后收款人将等待交易出现在首个区块中，然后在等到z个区块链接其后。此时，他仍然不能确切知道攻击者已经进展了多少个区块，但是假设诚实区块将耗费平均预期时间以产生一个区块，那么攻击者的潜在进展就是一个泊松分布，分布的期望值为：

$$\lambda = z\frac{q}{p}$$

当此情形，为了计算攻击者追赶上的概率，我们将攻击者取得进展区块数量的泊松分布的概率密度，乘以在该数量下攻击者依然能够追赶上的概率。

$$\sum_{k=0}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \begin{cases} (\frac{q}{p})^{(z-k)} & \text{if } k \le z \\ 1 & \text{if } k > z \end{cases}$$

化为如下形式，避免对无限数列求和：

$$1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \left(1 - \left(\frac{q}{p}\right)^{(z-k)}\right)$$

写为如下C语言代码：

`#include double AttackerSuccessProbability(double q, int z){double p = 1.0 - q;double lambda = z * (q / p);double sum = 1.0;int i, k;for (k = 0; k <= z; k++){double poisson = exp(-lambda);for (i = 1; i <= k; i++)poisson *= lambda / i;sum -= poisson * (1 - pow(q / p, z - k));}return sum;}` 对其进行运算，我们可以得到如下的概率结果，发现概率对z值呈指数下降。

当q=0.1时 z=0 P=1.0000000 z=1 P=0.2045873 z=2 P=0.0509779 z=3 P=0.0131722 z=4 P=0.0034552 z=5 P=0.0009137 z=6 P=0.0002428 z=7 P=0.0000647 z=8 P=0.0000173 z=9 P=0.0000046 z=10 P=0.0000012

当q=0.3时 z=0 P=1.0000000 z=5 P=0.1773523 z=10 P=0.0416605 z=15 P=0.0101008 z=20 P=0.0024804 z=25 P=0.0006132 z=30 P=0.0001522 z=35 P=0.0000379 z=40 P=0.0000095 z=45 P=0.0000024 z=50 P=0.0000006

求解令P<0.1%的z值：

为使P<0.001，则 q=0.10 z=5 q=0.15 z=8 q=0.20 z=11 q=0.25 z=15 q=0.30 z=24 q=0.35 z=41 q=0.40 z=89 q=0.45 z=340

12.结论

我们在此提出了一种不需要信用中介的电子支付系统。我们首先讨论了通常的电子货币的电子签名原理，虽然这种系统为所有权提供了强有力的控制，但是不足以防止双重支付。为了解决这个问题，我们提出了一种采用工作量证明机制的点对点网络来记录交易的公开信息，只要诚实的节点能够控制绝大多数的CPU计算能力，就能使得攻击者事实上难以改变交易记录。该网络的强健之处在于它结构上的简洁性。节点之间的工作大部分是彼此独立的，只需要很少的协同。每个节点都不需要明确自己的身份，由于交易信息的流动路径并无任何要求，所以只需要尽其最大努力传播即可。节点可以随时离开网络，而想重新加入网络也非常容易，因为只需要补充接收离开期间的工作量证明链条即可。节点通过自己的CPU计算力进行投票，表决他们对有效区块的确认，他们不断延长有效的区块链来表达自己的确认，并拒绝在无效的区块之后延长区块以表示拒绝。本框架包含了一个P2P电子货币系统所需要的全部规则和激励措施。

注释

1. W Dai（戴伟）,a scheme for a group of untraceable digital pseudonyms to pay each other with money and to enforce contracts amongst themselves without outside help（一种能够借助电子假名在群体内部相互支付并迫使个体遵守规则且不需要外界协助的电子现金机制）, “B-money”, http://www.weidai.com/bmoney.txt, 1998↵
2. H. Massias, X.S. Avila, and J.-J. Quisquater, “Design of a secure timestamping service with minimal trust requirements,”（在最小化信任的基础上设计一种时间戳服务器） In 20th Symposium on Information Theory in the Benelux, May 1999.↵
3. S. Haber, W.S. Stornetta, “How to time-stamp a digital document,” （怎样为电子文件添

加时间戳）In Journal of Cryptology, vol 3, No.2, pages 99-111, 1991.↵

4. D. Bayer, S. Haber, W.S. Stornetta, “Improving the efficiency and reliability of digital time-stamping,”（提升电子时间戳的效率和可靠性） In Sequences II: Methods in Communication, Security and Computer Science, pages 329-334, 1993.↵
5. S. Haber, W.S. Stornetta, “Secure names for bit-strings,”（比特字串的安全命名） In Proceedings of the 4th ACM Conference on Computer and Communications Security, pages 28-35, April 1997. on Computer and Communications Security, pages 28-35, April 1997.↵
6. A. Back, “Hashcash – a denial of service counter-measure,”（哈希现金——拒绝服务式攻击的克制方法）http://www.hashcash.org/papers/hashcash.pdf, 2002.↵
7. R.C. Merkle, “Protocols for public key cryptosystems,” （公钥密码系统的协议）In Proc. 1980 Symposium on Security and Privacy, IEEE Computer Society, pages 122-133, April 1980. S. Haber, W.S. Stornetta, “Secure names for bit-strings,”（比特字串安全命名） In Proceedings of the 4th ACM Conference on Computer and Communications Security, pages 28-35, April 1997. on Computer and Communications Security, pages 28-35, April 1997. H. Massias, X.S. Avila, and J.-J. Quisquater, “Design of a secure timestamping service with minimal trust requirements,”（在最小化信任的条件下设计一种时间戳服务器） In 20th Symposium on Information Theory in the Benelux, May 1999.↵
8. W. Feller, “An introduction to probability theory and its applications,” （概率学理论与应用导论）1957

附录2、交易脚本语言操作符，常量和符号

以下的表和描述参见https://en.bitcoin.it/wiki/Script

表1.脚本压入堆栈

符号	值 (十六进制)	描述
OP_0 or OP_FALSE	0x00	一个字节空串被压入堆栈中
1-75	0x01-0x4b	把接下来的N 个字节压入堆栈中，N 的取值在1 到75 之间
OP_PUSHDATA1	0x4c	下一个脚本字节包括N，会将接下来的N 个字节压入堆栈
OP_PUSHDATA2	0x4d	下两个脚本字节包括N，会将接下来的N 个字节压入堆栈
OP_PUSHDATA4	0x4e	下四个脚本字节包括N，会将接下来的N 个字节压入堆栈
OP_1NEGATE	0x4f	将脚本-1 压入堆栈
OP_RESERVED	0x50	终止- 交易无效（除非在未执行的OP_IF语句中）
OP_1 or OP_TRUE	0x51	将脚本1 压入堆栈
OP_2 to OP_16	0x52 to 0x60	将脚本N 压入堆栈，例如OP_2 压入脚本“2”

表2.有条件的流控制的操作符

符号	值 (十六进制)	描述
OP_NOP	0x61	无操作
OP_VER	0x62	终止- 交易无效（除非在未执行的OP_IF 语句中）
OP_IF	0x63	如果栈项元素值为0，语句将被执行
OP_NOTIF	0x64	如果栈项元素值不为0，语句将被执行
OP_VERIF	0x65	终止- 交易无效
OP_VERNOTIF	0x66	终止- 交易无效
OP_ELSE	0x67	如果前述的OP_IF 或OP_NOTIF 或OP_ELSE 未被执行，这些语句就会被执行
OP_ENDIF	0x68	终止OP_IF, OP_NOTIF, OP_ELSE 区块
OP_VERIFY	0x69	如果栈项元素值非真，则标记交易无效
OP_RETURN	0x6a	标记交易无效

表3.时间锁操作符

符号	值 (十六进制)	描述
OP_CHECKLOCKTIMEVERIFY (previously OP_NOP2)	0xb1	如果栈顶元素比交易锁定时间字段大，则将交易标记为无效。否则脚本评测将像OP_NOP操作一样继续执行。交易在一下4种之一的情况下是无效的：1.堆栈是空的；2.栈顶元素是负数；3.当交易锁定时间字段值少于500000000时，栈顶元素大于等于500000000，反之亦然；4.输入序列字段等于0xffffffff。具体内容详见BIP-65。
OP_CHECKSEQUENCEVERIFY (previously OP_NOP3)	0xb2	如果输入值(BIP 0068强制规定的顺序)的相对锁定时间不等于或多于栈顶元素值时，将交易标记为无效。具体内容详见BIP-112。

表4.堆栈操作符

符号	值 (十六进制)	描述
OP_TOALTSTACK	0x6b	从主堆栈中取出元素，推入辅堆栈。
OP_FROMALTSTACK	0x6c	从辅堆栈中取出元素，推入主堆栈
OP_2DROP	0x6d	移除栈顶两个元素
OP_2DUP	0x6e	复制栈顶两个元素
OP_3DUP	0x6f	复制栈顶三个元素
OP_2OVER	0x70	把栈底的第三、第四个元素拷贝到栈顶
OP_2ROT	0x71	移动第五、第六元素到栈顶
OP_2SWAP	0x72	将栈顶的两个元素进行交换
OP_IFDUP	0x73	如果栈项元素值不为0，复制该元素值
OP_DEPTH	0x74	Count the items on the stack and push the resulting count
OP_DROP	0x75	删除栈顶元素
OP_DUP	0x76	复制栈顶元素
OP_NIP	0x77	删除栈顶的下一个元素
OP_OVER	0x78	复制栈顶的下一个元素到栈顶
OP_PICK	0x79	把堆栈的第n 个元素拷贝到栈顶
OP_ROLL	0x7a	把堆栈的第n 个元素移动到栈顶
OP_ROT	0x7b	翻转栈顶的三个元素
OP_SWAP	0x7c	栈顶的三个元素交换
OP_TUCK	0x7d	拷贝栈顶元素并插入到栈顶第二个元素之后

表5.字符串接操作

符号	值 (十六进制)	描述
OP_CAT	0x7e	连接两个字符串，已禁用
OP_SUBSTR	0x7f	返回字符串的一部分，已禁用
OP_LEFT	0x80	在一个字符串中保留左边指定长度的子串，已禁用
OP_RIGHT	0x81	在一个字符串中保留右边指定长度的子串，已禁用
OP_SIZE	0x82	把栈顶元素的字符串长度压入堆栈

表6.二进制算术和条件

符号	值 (十六进制)	描述
OP_INVERT	0x83	所有输入的位取反，已禁用
OP_AND	0x84	对输入的所有位进行布尔与运算，已禁用
OP_OR	0x85	对输入的每一位进行布尔或运算，已禁用
OP_XOR	0x86	对输入的每一位进行布尔异或运算，已禁用
OP_EQUAL	0x87	如果输入的两个数相等，返回1，否则返回0
OP_EQUALVERIFY	0x88	与OP_EQUAL 一样，如结果为0，之后运行OP_VERIFY
OP_RESERVED1	0x89	终止- 无效交易（除非在未执行的OP_IF 语句中）
OP_RESERVED2	0x8a	终止-无效交易（除非在未执行的OP_IF 语句中）

表7.数值操作

符号	值 (十六进制)	描述
OP_1ADD	0x8b	栈顶值加1
OP_1SUB	0x8c	栈顶值减1
OP_2MUL	0x8d	无效（栈顶值乘2）
OP_2DIV	0x8e	无效（栈顶值除2）
OP_NEGATE	0x8f	栈顶值符号取反
OP_ABS	0x90	栈顶值符号取正
OP_NOT	0x91	如果栈顶值为0 或1，则输出1或0；否则输出0
OP_0NOTEQUAL	0x92	输入值为0 输出0；否则输出1
OP_ADD	0x93	弹出栈顶的两个元素，压入二者相加结果
OP_SUB	0x94	弹出栈顶的两个元素，压入二者相减（第二项减去第一项）结果
OP_MUL	0x95	禁用（栈顶两项的积）
OP_DIV	0x96	禁用（输出用第二项除以第一项的倍数）
OP_MOD	0x97	禁用（输出用第二项除以第一项得到的余数）
OP_LSHIFT	0x98	禁用（左移第二项，移动位数为第一项的二进制位数）
OP_RSHIFT	0x99	禁用（右移第二项，移动位数为第一项的二进制位数）

OP_BOOLAND	0x9a	布尔与运算，两项都不为0，输出1，否则输出0
OP_BOOLOR	0x9b	布尔或运算，两项有一个不为0，输出1，否则输出0
OP_NUMEQUAL	0x9c	两项相等则输出1，否则输出为0
OP_NUMEQUALVERIFY	0x9d	与NUMEQUAL 相同，如结果为0运行OP_VERIFY
OP_NUMNOTEQUAL	0x9e	如果栈顶两项不是相等数的话，则输出1
OP_LESSTHAN	0x9f	如果第二项小于栈顶项，则输出1
OP_GREATERTHAN	0xa0	如果第二项大于栈顶项，则输出1
OP_LESSTHANOREQUAL	0xa1	如果第二项小于或等于第一项，则输出1
OP_GREATERTHANOREQUAL	0xa2	如果第二项大于或等于第一项，则输出1
OP_MIN	0xa3	输出栈顶两项中较小的一项
OP_MAX	0xa4	输出栈顶两项中较大的一项
OP_WITHIN	0xa5	如果第三项的数值介于前两项之间，则输出1

表8.加密和散列操作

符号	值(十六进制)	描述
OP_RIPEMD160	0xa6	返回栈顶项的RIPEMD160 哈希值
OP_SHA1	0xa7	返回栈顶项SHA1 哈希值
OP_SHA256	0xa8	返回栈顶项SHA256 哈希值
OP_HASH160	0xa9	栈顶项进行两次HASH，先用SHA-256，再用RIPEMD-160
OP_HASH256	0xaa	栈顶项用SHA-256 算法HASH 两次
OP_CODESEPARATOR	0xab	标记已进行签名验证的数据
OP_CHECKSIG	0xac	交易所用的签名必须是哈希值和公钥的有效签名，如果为真，则返回1
OP_CHECKSIGVERIFY	0xad	与CHECKSIG 一样，但之后运行OP_VERIFY
OP_CHECKMULTISIG	0xae	对于每对签名和公钥运行CHECKSIG。所有的签名要与公钥匹配。因为存在BUG，一个未使用的外部值会从堆栈中删除。
OP_CHECKMULTISIGVERIFY	0xaf	与CHECKMULTISIG 一样，但之后运行OP_VERIFY

表9.非操作符

符号	值 (十六进制)	描述
OP_NOP1-OP_NOP10	0xb0-0xb9	无操作忽略

表10.仅供内部使用的保留关键字

符号	值 (十六进制)	描述
OP_SMALLDATA	0xf9	代表小数据域
OP_SMALLINTEGER	0xfa	代表小整数数据域
OP_PUBKEYS	0xfb	代表公钥域
OP_PUBKEYHASH	0xfd	代表公钥哈希域
OP_PUBKEY	0xfe	代表公钥域
OP_INVALIDOPCODE	0xff	代表当前未指定的操作码

附录3、比特币改进建议（BIPs）

比特币改进提案是向比特币社区提供信息的设计文档，或用于描述比特币的新功能，流程或环境。

根据BIP-01也就是BIP目的和指南（BIP Purpose and Guidelines）的规定，有三种BIP：

标准类BIP

描述影响大多数或所有比特币实现的任何更改，例如网络协议的更改，区块或交易有效性规则的更改，或影响使用比特币的应用程序的互操作性的任何更改或附加。

信息类BIP

描述比特币设计问题，或向比特币社区提供一般准则或信息，但不提出新功能。信息类BIP不一定代表比特币社区的共识或建议，因此用户和实施者可以忽略信息类BIP或遵循他们的建议。

过程类BIP

描述一个比特币过程，或者提出一个过程的更改（或一个事件）。过程类BIP类似于标准类BIP，但适用于比特币协议本身以外的其他领域。他们可能会提出一个实现，但不是比特币的代码库;他们经常需要社区的共识;与信息类BIP不同，它们不仅仅是建议，用户通常也不能随意忽略它们。例如包括程序，指南，决策过程的变化以及对比特币开发中使用的工具或环境的更改。任何元BIP也被视为一个过程BIP。

BIP记录在GitHub上的版本化存储库中：https://github.com/bitcoin/bips。 下表BIP的快照显示在2017年4月BIP的快照。了解有关现有BIP及其内容的最新信息请咨询权威机构。

BIP# Title Owner Type Status
BIP-1 BIP Purpose and Guidelines Amir Taaki Process Replaced
BIP-2 BIP process, revised Luke Dashjr Process Active
BIP-8 Version bits with guaranteed lock-in Shaolin Fry Informational Draft
BIP-9 Version bits with timeout and delay Pieter Wuille, Peter Todd, Greg Maxwell, Rusty Russell Informational Final
BIP-10 Multi-Sig Transaction Distribution Alan Reiner Informational Withdrawn
BIP-11 M-of-N Standard Transactions Gavin Andresen Standard Final
BIP-12 OP_EVAL Gavin Andresen Standard Withdrawn
BIP-13 Address Format for pay-to-script-hash Gavin Andresen Standard Final
BIP-14 Protocol Version and User Agent Amir Taaki, Patrick Strateman Standard Final
BIP-15 Aliases Amir Taaki Standard Deferred
BIP-16 Pay to Script Hash Gavin Andresen Standard Final
BIP-17 OP_CHECKHASHVERIFY (CHV) Luke Dashjr Standard Withdrawn
BIP-18 hashScriptCheck Luke Dashjr Standard Proposed
BIP-19 M-of-N Standard Transactions (Low SigOp) Luke Dashjr Standard Draft

BIP-20 URI Scheme Luke Dashjr Standard Replaced
BIP-21 URI Scheme Nils Schneider, Matt Corallo Standard Final
BIP-22 getblocktemplate - Fundamentals Luke Dashjr Standard Final
BIP-23 getblocktemplate - Pooled Mining Luke Dashjr Standard Final
BIP-30 Duplicate transactions Pieter Wuille Standard Final
BIP-31 Pong message Mike Hearn Standard Final
BIP-32 Hierarchical Deterministic Wallets Pieter Wuille Informational Final
BIP-33 Stratized Nodes Amir Taaki Standard Draft
BIP-34 Block v2, Height in Coinbase Gavin Andresen Standard Final
BIP-35 mempool message Jeff Garzik Standard Final
BIP-36 Custom Services Stefan Thomas Standard Draft
BIP-37 Connection Bloom filtering Mike Hearn, Matt Corallo Standard Final
BIP-38 Passphrase-protected private key Mike Caldwell, Aaron Voisine Standard Draft
BIP-39 Mnemonic code for generating deterministic keys Marek Palatinus, Pavol Rusnak, Aaron Voisine, Sean Bowe Standard Proposed
BIP-40 Stratum wire protocol Marek Palatinus Standard BIP number allocated BIP-41 Stratum mining protocol Marek Palatinus Standard BIP number allocated BIP-42 A finite monetary supply for Bitcoin Pieter Wuille Standard Draft
BIP-43 Purpose Field for Deterministic Wallets Marek Palatinus, Pavol Rusnak Informational Draft
BIP-44 Multi-Account Hierarchy for Deterministic Wallets Marek Palatinus, Pavol Rusnak Standard Proposed
BIP-45 Structure for Deterministic P2SH Multisignature Wallets Manuel Araoz, Ryan X. Charles, Matias Alejo Garcia Standard Proposed
BIP-47 Reusable Payment Codes for Hierarchical Deterministic Wallets Justus Ranvier Informational Draft
BIP-49 Derivation scheme for P2WPKH-nested-in-P2SH based accounts Daniel Weigl Informational Draft
BIP-50 March 2013 Chain Fork Post-Mortem Gavin Andresen Informational Final
BIP-60 Fixed Length "version" Message (Relay-Transactions Field) Amir Taaki Standard Draft
BIP-61 Reject P2P message Gavin Andresen Standard Final
BIP-62 Dealing with malleability Pieter Wuille Standard Withdrawn
BIP-63 Stealth Addresses Peter Todd Standard BIP number allocated BIP-64 getutxo message Mike Hearn Standard Draft
BIP-65 OP_CHECKLOCKTIMEVERIFY Peter Todd Standard Final
BIP-66 Strict DER signatures Pieter Wuille Standard Final
BIP-67 Deterministic Pay-to-script-hash multi-signature addresses through public key sorting Thomas Kerin, Jean-Pierre Rupp, Ruben de Vries Standard Proposed
BIP-68 Relative lock-time using consensus-enforced sequence numbers Mark Friedenbach,

BtcDrak, Nicolas Dorier, kinoshitajona Standard Final
BIP-69 Lexicographical Indexing of Transaction Inputs and Outputs Kristov Atlas Informational Proposed
BIP-70 Payment Protocol Gavin Andresen, Mike Hearn Standard Final
BIP-71 Payment Protocol MIME types Gavin Andresen Standard Final
BIP-72 bitcoin: uri extensions for Payment Protocol Gavin Andresen Standard Final
BIP-73 Use "Accept" header for response type negotiation with Payment Request URLs Stephen Pair Standard Final
BIP-74 Allow zero value OP_RETURN in Payment Protocol Toby Padilla Standard Draft
BIP-75 Out of Band Address Exchange using Payment Protocol Encryption Justin Newton, Matt David, Aaron Voisine, James MacWhyte Standard Draft
BIP-80 Hierarchy for Non-Colored Voting Pool Deterministic Multisig Wallets Justus Ranvier, Jimmy Song Informational Deferred
BIP-81 Hierarchy for Colored Voting Pool Deterministic Multisig Wallets Justus Ranvier, Jimmy Song Informational Deferred
BIP-83 Dynamic Hierarchical Deterministic Key Trees Eric Lombrozo Standard Draft
BIP-90 Buried Deployments Suhas Daftuar Informational Draft
BIP-99 Motivation and deployment of consensus rule changes ([soft/hard]forks) Jorge Timón Informational Draft
BIP-101 Increase maximum block size Gavin Andresen Standard Withdrawn
BIP-102 Block size increase to 2MB Jeff Garzik Standard Draft
BIP-103 Block size following technological growth Pieter Wuille Standard Draft
BIP-104 'Block75' - Max block size like difficulty t.khan Standard Draft
BIP-105 Consensus based block size retargeting algorithm BtcDrak Standard Draft
BIP-106 Dynamically Controlled Bitcoin Block Size Max Cap Upal Chakraborty Standard Draft
BIP-107 Dynamic limit on the block size Washington Y. Sanchez Standard Draft
BIP-109 Two million byte size limit with sigop and sighash limits Gavin Andresen Standard Rejected
BIP-111 NODE_BLOOM service bit Matt Corallo, Peter Todd Standard Proposed
BIP-112 CHECKSEQUENCEVERIFY BtcDrak, Mark Friedenbach, Eric Lombrozo Standard Final
BIP-113 Median time-past as endpoint for lock-time calculations Thomas Kerin, Mark Friedenbach Standard Final
BIP-114 Merkelized Abstract Syntax Tree Johnson Lau Standard Draft
BIP-120 Proof of Payment Kalle Rosenbaum Standard Draft
BIP-121 Proof of Payment URI scheme Kalle Rosenbaum Standard Draft
BIP-122 URI scheme for Blockchain references / exploration Marco Pontello Standard Draft
BIP-123 BIP Classification Eric Lombrozo Process Active
BIP-124 Hierarchical Deterministic Script Templates Eric Lombrozo, William Swanson

Informational Draft
BIP-125 Opt-in Full Replace-by-Fee Signaling David A. Harding, Peter Todd Standard Proposed
BIP-126 Best Practices for Heterogeneous Input Script Transactions Kristov Atlas Informational Draft
BIP-130 sendheaders message Suhas Daftuar Standard Proposed
BIP-131 "Coalescing Transaction" Specification (wildcard inputs) Chris Priest Standard Draft
BIP-132 Committee-based BIP Acceptance Process Andy Chase Process Withdrawn
BIP-133 feefilter message Alex Morcos Standard Draft
BIP-134 Flexible Transactions Tom Zander Standard Draft
BIP-140 Normalized TXID Christian Decker Standard Draft
BIP-141 Segregated Witness (Consensus layer) Eric Lombrozo, Johnson Lau, Pieter Wuille Standard Draft
BIP-142 Address Format for Segregated Witness Johnson Lau Standard Deferred
BIP-143 Transaction Signature Verification for Version 0 Witness Program Johnson Lau, Pieter Wuille Standard Draft
BIP-144 Segregated Witness (Peer Services) Eric Lombrozo, Pieter Wuille Standard Draft
BIP-145 getblocktemplate Updates for Segregated Witness Luke Dashjr Standard Draft
BIP-146 Dealing with signature encoding malleability Johnson Lau, Pieter Wuille Standard Draft
BIP-147 Dealing with dummy stack element malleability Johnson Lau Standard Draft
BIP-148 Mandatory activation of segwit deployment Shaolin Fry Standard Draft
BIP-150 Peer Authentication Jonas Schnelli Standard Draft
BIP-151 Peer-to-Peer Communication Encryption Jonas Schnelli Standard Draft
BIP-152 Compact Block Relay Matt Corallo Standard Draft
BIP-171 Currency/exchange rate information API Luke Dashjr Standard Draft
BIP-180 Block size/weight fraud proof Luke Dashjr Standard Draft
BIP-199 Hashed Time-Locked Contract transactions Sean Bowe, Daira Hopwood Standard Draft

附录4、隔离见证

隔离见证（segwit）是一次比特币共识规则和网络协议的升级，其提议和实施将基于BIP-9 软分叉方案，目前尚待激活（译者注：2017年8月24日，区块高度481,824，隔离见证正式激活）。

在密码学中，术语“见证”（witness）被用于形容一个加密难题的解决方案。在比特币中，“见证”满足了一种被放置在一个未使用的交易输出（unspent transaction output, UTXO）上的加密条件。

在比特币语境中，一个数字签名就是一种类型的“见证”（one type of witness）。但“见证”是一个更为广泛的任意解决方案，能够满足加诸于一个UTXO的条件，使UTXO解锁后可被花费。术语“见证”一词是一个更普遍用于“解锁脚本”（或scriptSig）的术语。

在引入“隔离见证”之前，每一个交易输入后面都跟着用来对其解锁的见证数据，见证数据作为输入的一部分被内嵌其中。术语“隔离见证”（ segregated witness），或简称为“segwit”，简单理解就是将某个特定输出的签名分离开，或将某个特定输入的脚本进行解锁。用最简单的形式来理解就是“分离解锁脚本”（separate scriptSig），或“分离签名”（separate signature）

因此，隔离见证就是比特币的一种结构性调整，旨在将见证数据部分从一笔交易的scriptSig（解锁脚本）字段移出至一个伴随交易的单独的见证数据结构。客户端请求交易数据时可以选择要或不要该部分伴随的见证数据。

在这一章节，我们将会看到隔离见证的一些好处，描述用于部署和实施该结构性调整的机制，并展示隔离见证在交易和地址中的运用。

隔离见证由以下BIPs定义：

BIP-141隔离见证的主要定义

BIP-143版本0见证程序的交易签名验证

BIP-144对等服务——新的网络消息和序列化格式

BIP-145隔离见证（对于矿工）的 getblocktemplate 升级

4.1为什么需要隔离见证？

隔离见证是一个将在多方面产生影响的结构性调整——可扩展性、安全性、经济刺激以及比特币整体性能：

交易延展性

将见证移出交易后，用作标识符的交易哈希不再包含见证数据。因为见证数据是交易中唯一可被第三方修改（参见 交易识别符 章节）的部分，移除它的同时也移除了交易延展性攻击的机会。通过隔离见证，交易变得对任何人（创建者本人除外）都不可变，这极大地提高了许多其它依赖于高级比特币交易架构的协议的可执行性。比如支付通道、跨连交易和闪电网络。

脚本版本管理

在引入隔离见证脚本后，类似于交易和区块都有其版本号，每一个锁定脚本前也都有了一个脚本版本号。脚本版本号的条件允许脚本语言用一种向后兼容的方式（也就是软分叉升级）升级，以引入新的脚本操作数、语法或语义。非破坏性升级脚本语言的能力将极大地加快比特币的创新速度。

网络和存储扩展

见证数据通常是交易总体积的重要贡献者。更复杂的脚本通常非常大，比如那些用于多重签名或支付通道的脚本。有时候这些脚本占据了一笔交易的大部分（超过75%）空间。通过将见证数据移出交易，隔离见证提升了比特币的可扩展性。节点能够在验证签名后去除见证数据，或在作简单支付验证时整个忽略它。见证数据不需要被发送至所有节点，也不需要被所有节点存储在硬盘中。

签名验证优化

隔离见证升级签名函数（CHECKSIG, CHECKMULTISIG, 等）减少了算法的计算复杂性。引入隔离见证前，用于生成签名的算法需要大量的哈希操作，这些操作与交易的大小成正比。在O(n2)中关于签名操作数量方面，数据哈希计算增加，在所有节点验证签名上引入了大量计算负担。引入隔离见证后，算法更改减少了O(n2)的复杂性。

离线签名改进

隔离见证签名包含了在被签名的哈希散列中，每个输入所引用的值（数量）。在此之前，一个离线签名装置，比如硬件钱包，必须在签署交易前验证每一个输入的数量。这通常是通过大量的数据流来完成的，这些数据是关于以前的交易被引用作为输入的。由于该数量现在是已签名的承诺哈希散列的一部分，因此离线装置不需要以前的交易。如果数量不匹配（被一个折中的在线系统误报），则签名无效。

4.2隔离见证如何工作

乍一看，隔离见证似乎是对交易如何构建的更改，因此是一个交易层面的特性，但事实并非如此。实际上，隔离见证也更改了单个UTXO如何被使用的方式，因此它是一个输出层面的特性。

一个交易可以引用隔离见证输出或传统（内联见证）输出，或者两者皆可。因此，把一个交易称作“隔离见证交易”是没有意义的。但是我们可以把某个特定的交易输出叫做“隔离见证输出”。

当一个交易引用一个UTXO，它必须提供一个见证。如果是传统的UTXO，一个交易在引用它时，UTXO的锁定脚本要求见证数据在该交易输出部分中以“内联”（inline）的方式被提供。但隔离见证UTXO指定的锁定脚本却能满足处于输入之外（被隔离）的见证数据。

4.3软分叉（向后兼容性）

隔离见证对于输出和交易的构建的方式是一个十分重大的改变。这样的改变将通常需要每一个比特币节点和钱包同时发生，以改变共识规则——即所谓的“硬分叉”。但是，隔离见证通过一个更少破坏性的改变引入，这种变化能向后兼容，被称作“软分叉”。这种类型的升级允许未升级的软件去忽略那些改变然后继续去操作避免任何分裂。

隔离见证输出被设计成老的“非隔离见证”系统仍然能够验证它们，对于老的钱包或节点来说，一个隔离见证输出看起来就像一个“任何人都能花费”（anyone can spend）的输出。这样的输出能被一个空的签名花费，因此一个交易里面没有签名（签名被隔离）的事实也并不会导致该交易不被验证。但是，更新的钱包和挖矿节点能够看到隔离见证输出，并期望在交易的见证数据中为该输出找到一个有效的见证。

4.4隔离见证输出和交易示例

让我们来看一些交易示例，看看他们是如何随着隔离见证而改变的。我们将首先看看通过隔离见证程序，如何被改变一个“支付给公钥哈希”（Pay-to-Public-Key-Hash ，P2PKH）的支付。然后，我们再看同样的隔离见证如何作用于“支付给脚本公钥”（Pay-to-Script-Hash ，P2SH）脚本。最后，我们会看看以上两种隔离见证程序如何可以被内嵌入一个 P2SH 脚本。

4.4.1Pay-to-Witness-Public-Key-Hash (P2WPKH)

在【一杯咖啡】的例子中，Alice为一杯咖啡创建了一笔交易去付款给Bob，该笔交易构建了一个价值0.015BTC的 P2PKH 输出（Bob可用来花费），该输出脚本看起来像这样：

P2PKH 输出脚本示例：

```
DUP HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 EQUALVERIFY CHECKSIG
```

如果通过隔离见证，Alice将会创建一个“支付给见证公钥哈希”（P2WPKH）脚本，看起来是这样的：

P2WPKH 输出脚本示例：

```
0 ab68025513c3dbd2f7b92a94e0581f5d50f654e7
```

正如你所见，一个隔离见证输出的锁定脚本相对一个传统输出明显大为简化。它包含两个值，这些值被推送到脚本评估堆栈中。对于一个传统（非隔离见证）比特币客户端来说，这两个推送值看起来像一个任何人都能花费的输出而不需要签名（或者更确切的说，能被空的签名使用）。而对一个更新的、隔离见证客户端来说，第一个数字（0）被解释为一个版本号（见证版本），第二部分（20字节）相当于一个锁定脚本，被称为“见证程序”（ witness program）。这20字节的见证程序即是公钥哈希值，就像在 P2PKH 脚本中一样。

现在，让我们来看看Bob用来去花费这个输出相应的交易。对于初始脚本（非隔离见证），Bob的交易必须包含签名在交易输入中：

以下被解码的交易显示了一个 P2PKH 输出被一个签名使用：

```
 [...]“Vin” : ["txid":
"0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2","vout": 0, "scriptSig":
“<Bob’s scriptSig>”,][...]
```

但是，使用一个隔离见证输出时，交易输入内不存在签名。替代的，Bob的交易只有一个空的scriptSig ，并在交易本身之外包含了一个隔离见证：

以下被解码的交易显示了一个被隔离见证数据使用的 P2WPKH 输出：

```
 [...]“Vin” : ["txid":
"0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2","vout": 0, "scriptSig":
“”,][...]“witness”: “<Bob’s witness data>”[...]
```

4.4.2钱包的P2WPKH (Pay-to-Witness-Script-Hash)构造

尤其值得注意的是，P2WPKH 应该只有收款人（接收方）创建，而不是由发送者从已知的公钥、P2PKH 脚本或地址进行转换。发送方无从知道接收者的钱包是否具有能力构建隔离见证交易，并使用 P2WPKH 输出。

另外，P2WPKH 输出必须从被压缩的公钥的哈希值中创建。未压缩公钥在隔离见证中是非标准的，可能会被将来的软分叉明确禁用。如果在 P2WPKH 中使用的哈希值来自未压缩的公钥，那么它可能不可用，您将可能丢失资金。P2WPKH 输出应该由收款人的钱包，通过从钱包私钥中获取压缩公钥进行创建。

警告 P2WPKH 应该由收款人（接收者）通过将被压缩的公钥转换成P2WPKH哈希值进行创建。你绝不应该将P2PKH脚本、比特币地址或未压缩公钥转换成P2WPKH见证脚本。

4.4.3Pay-to-Witness-Script-Hash (P2WSH)

第二种类型的检证程序对应一个“支付给脚本哈希”（ Pay-to-Script-Hash , P2SH）脚本。我们在[p2sh]这章中见过。在哪个例子中，穆罕默德的公司使用了P2SH 来表达一个多重签名脚本。对穆罕默德的公司的支付被编码成一个这样的锁定脚本：

P2SH 输出脚本示例：

```
HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e EQUAL
```

这个P2SH脚本引用了一个赎回脚本（ redeem script ）的哈希值，改脚本定义了一个“2 of 3”的多重签名需求来使用资金。为了使用该输出，穆罕默德的公司将提供这个赎回脚本（其哈希值与P2SH 输出中的脚本哈希值匹配），以及满足该赎回脚本所需的签名，所有这些都在交易输出中：

显示一个P2SH输出被使用的解码交易：

```
 [...]“Vin” : ["txid": "abcdef12345...","vout": 0, "scriptSig": “<SigA> <SigB> <2 PubA PubB PubC PubD PubE 5 CHECKMULTISIG>”,]
```

现在，让我们看看整个示例如何升级成为隔离见证。如果穆罕默德的客户使用的是一个隔离见证兼容的钱包，他们就会付款，创建一个“支付给脚本哈希”(P2WSH)输出，看起来就像这样：

P2WSH 输出脚本示例：

```
0 9592d601848d04b172905e0ddb0adde59f1590f1e553ffc81ddc4b0ed927dd73
```

再一次，就像P2WPKH的例子一样，你可以看到，隔离见证等同脚本要简单得多，省略了各种你在P2SH脚本中见到的脚本操作符。相反，隔离见证程序仅包含两个推送到堆栈的值：一个见证版本（0）和一个32字节的赎回脚本的哈希值。

提示 当P2SH使用20字节的RIPEMD160(SHA256(script)) 哈希值时，P2WSH见证程序使用了一个32字节的SHA256（脚本）哈希值。在选择哈希算法时，这一差异是有意为之，被用于通过哈希值长度来区分两种类型的见证程序（P2WPKH and P2WSH），并为P2WSH（128位而不是 80 位P2SH）提供更强的安全性。

穆罕默德的公司可以通过提供正确的赎回脚本和足够的签名满足并花出P2WSH输出。作为见证数据的一部分，赎回脚本和签名被隔离在此支出交易之外。在交易输入内部，穆罕默德的钱包会防止一个空的scriptSig：

显示了一个P2WSH输出被用隔离见证数据花出的解码交易：

```
 [...]“Vin” : ["txid": "abcdef12345...","vout": 0, "scriptSig": “”,][...]“witness”: “<SigA> <SigB> <2 PubA PubB PubC PubD PubE 5 CHECKMULTISIG>”[...]
```

4.4.4区分P2WPKH和P2WSH

在前面的两节中，我们展示了两种类型的检证程序：支付给见证公钥哈希 (P2WPKH)和 支付给见证脚本哈希(P2WSH) 。这两种检证程序都有一个字节版本号和一个跟随其后的更长的哈希值组成。它们看起来非常相似，但是被解释得非常不同：一个被解释为一个公钥哈希值，它被一个签名所满足，另一个被解释为一个脚本哈希值，它被一个赎回脚本所满足。他们之间的关键区别是哈希值的长度：

- P2WPKH中的公钥哈希值是20字节。

- P2WSH中的脚本哈希值是32字节。

这个区别使得钱包可以对这两种类型的见证程序进行区分。通过查看哈希值的长度，钱包可以确定它是什么类型的见证程序，P2WPKH 或者 P2WSH。

4.5隔离见证升级

正如我们前面看到的例子，隔离见证的升级需要经过两步过程。首先，钱包必须创建特殊的隔离型输出。然后，这些输出可以被知道如何构建隔离见证交易的钱包花费。在这些例子中，Alice的钱包是segwit意识到的，并且能够使用Segregated Witness脚本创建特殊输出。Bob的钱包也是segwit意识到，并能够花这些输出。从这个例子中可能不明显的是，在实践中，Alice的钱包需要知道Bob使用了一个支持segwit的钱包，并可以使用这些输出。否则，如果Bob的钱包没有升级，并且Alice试图对Bob进行隔离见证付款，那么Bob的钱包将无法检测到这些付款。

提示 对于P2WPKH和P2WSH付款类型，发件人和收件人钱包都需要升级才能使用segwit。此外，发件人的钱包需要知道收件人的钱包是否具有隔离识别功能。

隔离见证不会在整个网络中同时实施。相反，隔离见证被实施为向后兼容的升级，其中新老客户可以共存。钱包开发人员将独立升级钱包软件以添加隔离区功能。当发件人和收件人都可以感知到网志时，使用P2WPKH和P2WSH付款类型。传统的P2PKH和P2SH将继续为非升级的钱包工作。这留下了两个重要的情况，下一节将讨论这个情况：

- 发件人的钱包的能力，不是兼容segwit到付款的收件人的钱包，可以处理segwit交易。
- 具有隔离识别功能的发件人钱包能够识别和区分具有隔离识别功能的收件人和不是他们地址的收件人。

4.5.1在P2SH中嵌入隔离见证

举个例子，假设Alice的钱包没有升级到segwit，但是Bob的钱包已经升级，可以处理segwit交易。Alice和Bob可以使用“旧”非segwit交易。但是Bob很可能会使用segwit来降低交易费用，利用适用于见证数据的折扣。

在这种情况下，Bob的钱包可以构建一个包含一个segwit脚本的P2SH地址。Alice的钱包认为这是一个“正常的”P2SH地址，并可以在没有任何segwit的知识的情况下付款。然后，Bob的钱包可以通过隔离交易来支付这笔款项，充分利用隔离交易并降低交易费用。

两种形式的见证脚本P2PKH和P2WSH都可以嵌入到P2SH地址中。第一个是P2SH（P2PKH），第二个是P2SH（P2WSH）。

4.5.2在 P2SH 中的 P2WPKH

见证脚本的第一种形式是P2SH（P2WPKH）。这是一个Pay-to-Witness-Public-Key-Hash证明程序，嵌入在Pay-to-Script-Hash脚本中，所以它可以被不知道segwit的钱包使用。

Bob的钱包用Bob的公钥构造了一个P2WPKH见证程序。这个见证程序然后被散列，结果散列被编码为P2SH脚本。P2SH脚本被转换成比特币地址，一个以“3”开始的地址，正如我们在[P2SH]部分看到的那样。

Bob的钱包从我们之前看到的P2WPKH 见证程序开始：

Bob的见证程序：

```
0 ab68025513c3dbd2f7b92a94e0581f5d50f654e7
```

P2WPKH见证程序由见证版本和Bob的20字节公钥散列组成。

Bob的钱包然后散列之前的见证程序，先用SHA256，然后用RIPEMD160，产生另一个20字节的散列：

P2WPKH见证程序的HASH160

```
3e0547268b3b19288b3adef9719ec8659f4b2b0b
```

然后见证程序的hash嵌入到P2SH脚本中：

```
HASH160 3e0547268b3b19288b3adef9719ec8659f4b2b0b EQUAL
```

最后，P2SH脚本转换为P2SH比特币地址：

```
37Lx99uaGn5avKBxiW26HjedQE3LrDCZru
```

现在，Bob可以显示这个地址给顾客付钱买咖啡。Alice的钱包可以支付给不支持隔离见证的地址，就像任何其他比特币地址一样。尽管Alice的钱包不支持segwit，但它创建的付款可以由Bob使用segwit交易进行支付。

4.5.3P2SH内的P2WSH

同样，一个用于multisig脚本或其他复杂脚本的P2WSH见证程序可以嵌入到P2SH脚本和地址中，使得任何钱包都可以进行与segwit兼容的支付。

正如我们在Pay-to-Witness-Script-Hash（P2WSH）中看到的，穆罕默德的公司正在使用隔离见证对多重签名脚本的付款。为了让任何客户支付他的公司，无论他们的钱包是否升级为隔离开关，穆罕默德的钱包都可以在P2SH脚本中嵌入P2WSH见证程序。

首先，穆罕默德的钱包创建与多重签名脚本对应的P2WSH见证程序，并用SHA256进行哈希处理：

穆罕默德的钱包创建了P2WSH见证程序

```
0 9592d601848d04b172905e0ddb0adde59f1590f1e553ffc81ddc4b0ed927dd73
```

然后，见证程序本身用SHA256和RIPEMD160散列，产生一个新的20字节散列，如传统的P2SH所使用的散列：

P2WSH见证程序的HASH160

`86762607e8fe87c0c37740cddee880988b9455b2`

接下来，穆罕默德的钱包将哈希码放入P2SH脚本中：

`HASH160 86762607e8fe87c0c37740cddee880988b9455b2 EQUAL`

最后，钱包从这个脚本构造一个比特币地址：

```
3Dwz1MXhM6EfFoJChHCxh1jWHb8GQqRenG
```

现在，穆罕默德的客户可以付款到这个地址，而不需要支持segwit。然后，穆罕默德的公司可以构建隔离交易来支付这些款项，利用包括较低的交易费用在内的隔离功能。

4.5.4隔离见证地址

在将比特币部署在比特币网络之后，钱包升级之前需要一些时间。因此，很可能像我们在前一节中看到的那样，segwit将主要用于嵌入到P2SH中，至少几个月。

但是，最终几乎所有的钱包都能够支持隔离支付。那时就不再需要在P2SH中嵌入segwit。因此，可能会创建一种新的比特币地址形式，表明接收者是具有隔离意识的，并直接对证人程序进行编码。有关隔离见证人地址计划的建议有很多，但没有一个被积极推行。

4.5.5交易标识符

隔离见证的最大好处之一就是消除了第三方交易延展性。

在segwit之前，交易可以通过第三方微妙地修改其签名，在不改变任何基本属性（输入，输出，金额）的情况下更改其交易ID（散列）。这为拒绝服务攻击创造了机会，以及攻击了编写不好的钱包软件，这些软件假定未经证实的交易哈希是不可变的。

通过引入隔离见证，交易有两个标识符txid和wtxid。传统的交易ID txid是序列化交易的双SHA256散列，没有见证数据。交易wtxid是具有见证数据的交易的新序列化格式的双SHA256散列。

传统txid的计算方式与nonsegwit交易完全相同。但是，由于segwit交易在每个输入中都有空的scriptSigs，因此没有可由第三方修改的部分交易。因此，在隔离交易中，即使交易未经确认，txid也是第三方不可改变的。

wtxid就像一个“扩展的”ID，因为hash也包含了见证数据。如果交易没有见证数据传输，则wtxid和txid是相同的。注意，由于wtxid包含见证数据（签名），并且由于见证数据可能具有延展性，所以在交易确认之前，wtxid应该被认为是可延展的。只有当交易的输入是segwit输

入时，第三方才可以认定segwit交易的txid不可变。

提示 隔离见证交易有两个ID：txid和wtxid。txid是没有见证数据的交易的散列，wtxid是包含见证数据的散列。所有输入为隔离开关输入的交易不受第三方交易延展性影响。

4.6隔离见证新的签名算法

隔离见证修改了四个签名验证函数（CHECKSIG，CHECKSIGVERIFY，CHECKMULTISIG和CHECKMULTISIGVERIFY）的语义，改变了交易承诺散列的计算方式。

比特币交易中的签名应用于交易哈希，交易数据计算，锁定数据的特定部分，表明签名者对这些值的承诺。例如，在简单的SIGHASH_ALL类型签名中，承诺哈希包括所有的输入和输出。

不幸的是，计算承诺哈希的方式引入了验证签名的节点可能被迫执行大量哈希计算的可能性。具体而言，散列运算相对于交易中的签名操作的数量增加O（n^2）。因此，攻击者可以通过大量的签名操作创建一个交易，导致整个比特币网络不得不执行数百或数千个哈希操作来验证交易。

Segwit代表了通过改变承诺散列计算方式来解决这个问题的机会。对于segwit版本0见证程序，使用BIP-143中规定的改进的承诺哈希算法进行签名验证。

新算法实现了两个重要目标。首先，散列操作的数量比签名操作的数量增加了一个更加渐进的O（n），减少了用过于复杂的交易创建拒绝服务攻击的机会。其次，承诺散列现在还包括作为承诺的一部分的每个输入的值（金额）。这意味着签名者可以提交特定的输入值，而不需要“获取”并检查输入引用的前一个交易。在离线设备（如硬件钱包）的情况下，这极大地简化了主机与硬件钱包之间的通信，消除了对以前的交易流进行验证的需要。硬件钱包可以接受不可信主机“输入”的输入值。由于签名是无效的，如果输入值不正确，硬件钱包在签名输入之前不需要验证该值。

4.7隔离见证的经济激励

比特币挖掘节点和完整节点会产生用于支持比特币网络和区块链的资源的成本。随着比特币交易量的增加，资源成本（CPU，网络带宽，磁盘空间，内存）也在增加。矿工通过与每次交易的大小（字节）成比例的费用来补偿这些成本。不挖矿的完整的节点没有得到补偿，所以他们承担这些成本，因为他们需要运行一个权威的充分验证全索引节点，可能是因为他们使用节点操作比特币业务。

如果没有交易费用，比特币数据的增长可能会大幅增加。费用旨在通过基于市场的价格发现机制，使比特币用户的需求与交易对网络带来的负担相一致。

基于交易规模的费用计算将交易中的所有数据视为相同的成本。但是从完整节点和矿工的角度来看，交易的某些部分的成本要高得多。加入比特币网络的每笔交易都会影响节点上四种资源的消耗：

- 磁盘空间

每笔交易都存储在区块链中，并添加到区块链的总大小中。区块链存储在磁盘上，但可以通过“修剪”较旧的交易来优化存储。

- CPU

每个交易都必须经过验证，这需要CPU时间。

- 带宽

每笔交易至少通过网络传输一次（通过泛洪传播）。如果在块传播协议中没有任何优化，交易将作为块的一部分再次传输，从而将对网络容量的影响加倍。

- 内存

验证交易的节点将UTXO索引或整个UTXO设置在内存中，以加速验证。由于内存至少比磁盘贵一个数量级，所以UTXO集的增长对运行节点的成本贡献不成比例。

从列表中可以看出，并不是交易的每个部分都对运行节点的成本或者比特币支持更多交易的能力产生同等的影响。交易中最昂贵的部分是新创建的输出，因为它们被添加到内存中的UTXO集。相比之下，签名（又名见证数据）为网络增加了最小的负担，并且节省了运行节点的成本，因为见证数据只被验证一次，之后又不再使用。此外，在收到新的交易并验证见证数据之后，节点可以立即丢弃该见证数据。如果按照交易规模计算费用，而不区分这两种数据，那么市场上的收费激励就不符合交易实际成本。事实上，目前的费用结构实际上鼓励了相反的行为，因为见证数据是交易的最大部分。

费用所产生的激励因为它们影响钱包的行为。所有的钱包都必须实行一些策略来组合交易，这些策略要考虑到隐私（减少地址重复使用），碎片化（大量松动）以及收费等因素。如果费用压倒性地促使钱包在交易中使用尽可能少的投入，这可能导致UTXO选择和改变地址策略，从而不经意地膨胀UTXO集合。

交易在其输入中消耗UTXO，并用它们的输出创建新的UTXO。因此，交易输入比输出多将导致UTXO集合的减少，而输出多于输入的交易将导致UTXO集合的增加。让我们考虑输入和输出之间的差异，并称之为“Net-new-UTXO”。这是一个重要的指标，因为它告诉我们一个交易对最昂贵的全网资源，内存 UTXO设置。正值Net-new-UTXO交易增加了这一负担。负值Net-new-UTXO交易减轻了负担。因此，我们希望鼓励交易是负Net-new-UTXO或零Net-new-UTXO的中值。

让我们来看一个例子，说明有无隔离见证交易费用计算产生了哪些激励措施。我们将看两个不同的交易。交易A是3输入，2输出的交易，其具有-1的净新UTXO度量，这意味着它消耗比它创建的多一个UTXO，将UTXO减1。交易B是2输入3输出的交易，其具有1的净新UTXO度

量，这意味着它向UTXO集增加一个UTXO，对整个比特币网络施加额外的成本。这两个交易都使用多重签名（2/3）脚本来演示复杂脚本如何增加隔离证人对费用的影响。我们假设交易费为每字节30 satoshi，证据数据为75％的折扣费用：

- 未使用隔离见证

  交易费用：25,710 satoshi 交易B费用：18,990 satoshi

- 使用隔离见证

  交易手续费：8,130 satoshi 交易B手续费：12,045 satoshi

当隔离见证实施时，这两个交易都较为便宜。但是比较这两笔交易的成本，我们发现在隔离见证之前，交易净收益为负的净新UTXO的收费更高。在隔离见证之后，交易费用与最小化新的UTXO创造的激励相一致，而不会无意中惩罚许多输入的交易。

因此，隔离见证对比特币用户支付的费用有两个主要的影响。首先，segwit通过折扣见证数据和增加比特币区块链的能力来降低交易的总体成本。其次，segwit对证人数据的折扣纠正了可能无意中在UTXO集合中产生更多膨胀的激励的错位。

Bitcore是BitPay提供的一套工具。 其目标是为Bitcoin开发人员提供易于使用的工具。 几乎所有的Bitcore的代码都是用JavaScript编写的。 有一些专门为NodeJS编写的模块。 最后，Bitcore的“节点”模块包括Bitcoin Core的C ++代码。 有关详细信息，请参阅https://bitcore.io。

Bitcore的功能列表

Bitcoin full node (bitcore-node)
Block explorer (insight)
Block, transaction, and wallet utilities (bitcore-lib)
Communicating directly with Bitcoin's P2P network (bitcore-p2p)
Seed entropy mnemonic generation（种子熵助记符） (bitcore-mnemonic)
Payment protocol (bitcore-payment-protocol)
Message verification and signing (bitcore-message)
Elliptic curve Integrated Encryption Scheme（椭圆曲线综合加密方案） (bitcore-ecies)
Wallet service (bitcore-wallet-service)
Wallet client (bitcore-wallet-client)
Playground (bitcore-playground)
Integrating services directly with Bitcoin Core (bitcore-node)

Bitcore库示例

先决条件

NodeJS >= 4.x 或者使用hosted online playground

如果使用NodeJS和节点REPL：

```
$ npm install -g bitcore-lib bitcore-p2p
$ NODE_PATH=$(npm list -g | head -1)/node_modules node
```

使用bitcore-lib的钱包示例

使用关联的私钥创建新的比特币地址：

```
> bitcore = require('bitcore-lib')
> privateKey = new bitcore.PrivateKey()
> address = privateKey.toAddress().toString()
```

创建分层确定性私钥和地址：

```
> hdPrivateKey = bitcore.HDPrivateKey()
> hdPublicKey = bitcore.HDPublicKey(hdPrivateKey)
> hdAddress = new bitcore.Address(hdPublicKey.publicKey).toString()
```

从UTXO创建和签署交易：

```
> utxo = {
  txId: transaction id containing an unspent output,
  outputIndex: output indexi e.g. 0,
  address: addressOfUtxo,
  script: bitcore.Script.buildPublicKeyHashOut(addressOfUtxo).toString(),
  satoshis: amount sent to the address
}
> fee = 3000 //set appropriately for conditions on the network
> tx = new bitcore.Transaction()
        .from(utxo)
        .to(address, 35000)
        .fee(fee)
        .enableRBF()
        .sign(privateKeyOfUtxo)
```

替换mempool中的最后一个交易（替换费）：

```
> rbfTx = new Transaction()
           .from(utxo)
           .to(address, 35000)
           .fee(fee*2)
           .enableRBF()
           .sign(privateKeyOfUtxo);
> tx.serialize();
> rbfTx.serialize();
```

将交易广播到比特币网络（注意：仅广播有效交易;请参阅https://bitnodes.21.co/nodes）：
将以下代码复制到名为broadcast.js的文件中。
tx和rbfTx变量分别是tx.serialize（）和rbfTx.serialize（）的输出。
为了更换费用，对等人必须支持bitcoind选项mempoolreplace并将其设置为1。
运行文件节点broadcast.js：

```
var p2p = require('bitcore-p2p');
var bitcore = require('bitcore-lib');
var tx = new bitcore.Transaction('output from serialize function');
var rbfTx = new bitcore.Transaction('output from serialize function');
var host = 'ip address'; //use valid peer listening on tcp 8333
var peer = new p2p.Peer({host: host});
var messages = new p2p.Messages();
peer.on('ready', function() {
  var txs = [messages.Transaction(tx), messages.Transaction(rbfTx)];
  var index = 0;
  var interval = setInterval(function() {
    peer.sendMessage(txs[index++]);
    console.log('tx: ' + index + ' sent');
    if (index === txs.length) {
      clearInterval(interval);
      console.log('disconnecting from peer: ' + host);
      peer.disconnect();
    }
  }, 2000);
});
peer.connect();
```

附录6、pycoin

最初由Richard Kiss编写和维护的Python库pycoin是一个基于Python的库，支持对比特币密钥和交易进行操作，甚至支持足够的脚本语言来适当地处理非标准交易。

pycoin库支持Python 2（2.7.x）和Python 3（3.3之后），并附带一些方便的命令行实用程序ku和tx。

1.实用工具（KU）

命令行实用程序ku（“密钥实用程序”）是用于操纵密钥的瑞士军刀。 它支持BIP-32键，WIF和地址（比特币和代币）。 以下是一些例子。

使用GPG和/ dev / random的默认熵源创建一个BIP-32密钥：

```
$ ku create


input           : create
network         : Bitcoin
wallet key      : xprv9s21ZrQH143K3LU5ctPZTBnb9kTjA5Su9DcWHvXJemiJBsY7VqXUG7hipgdWaU
                    m2nhnzdvxJf5KJo9vjP2nABX65c5sFsWsV8oXcbpehtJi
public version  : xpub661MyMwAqRbcFpYYiuvZpKjKhnJDZYAkWSY76JvvD7FH4fsG3Nqiov2CfxzxY8
                    DGcpfT56AMFeo8M8KPkFMfLUtvwjwb6WPv8rY65L2q8Hz
tree depth      : 0
fingerprint     : 9d9c6092
parent f'print  : 00000000
child index     : 0
chain code      : 80574fb260edaa4905bc86c9a47d30c697c50047ed466c0d4a5167f6821e8f3c
private key     : yes
secret exponent : 112471538590155650688604752840386134637231974546906847202389294096567806844862
 hex            : f8a8a28b28a916e1043cc0aca52033a18a13cab1638d544006469bc171fddfbe
wif             : L5Z54xi6qJusQT42JHA44mfPVZGjyb4XBRWfxAzUWwRiGx1kV4sP
 uncompressed   : 5KhoEavGNNH4GHKoy2Ptu4KfdNp4r56L5B5un8FP6RZnbsz5Nmb
public pair x   : 76460638240546478364843397478278468101877117767873462127021560368290114016034
public pair y   : 59807879657469774102040120298272207730921291736633247737077406753676825777701
 x as hex       : a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
 y as hex       : 843a0f6ed9c0eb1962c74533795406914fe3f1957c5238951f4fe245a4fcd625
y parity        : odd
key pair as sec : 03a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
 uncompressed   : 04a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
                    843a0f6ed9c0eb1962c74533795406914fe3f1957c5238951f4fe245a4fcd625
hash160         : 9d9c609247174ae323acfc96c852753fe3c8819d
 uncompressed   : 8870d869800c9b91ce1eb460f4c60540f87c15d7
Bitcoin address : 1FNNRQ5fSv1wBi5gyfVBs2rkNheMGt86sp
 uncompressed   : 1DSS5isnH4FsVaLVjeVXewVSpfqktdiQAM
```

从密码短语创建一个BIP-32密钥：

警告 这个例子中的密码很容易猜到。

```
$ ku P:foo


input           : P:foo
network         : Bitcoin
wallet key      : xprv9s21ZrQH143K31AgNK5pyVvW23gHnkBq2wh5aEk6g1s496M8ZMjxncCKZKgb5j
                    ZoY5eSJMJ2Vbyvi2hbmQnCuHBujZ2WXGTux1X2k9Krdtq
public version  : xpub661MyMwAqRbcFVF9ULcqLdsEa5WnCCugQAcgNd9iEMQ31tgH6u4DLQWoQayvtS
                    VYFvXz2vPPpbXE1qpjoUFidhjFj82pVShWu9curWmb2zy
tree depth      : 0
fingerprint     : 5d353a2e
parent f'print  : 00000000
child index     : 0
chain code      : 5eeb1023fd6dd1ae52a005ce0e73420821e1d90e08be980a85e9111fd7646bbc
private key     : yes
secret exponent : 65825730547097305716057160437970790220123864299761908948746835886007
793998275
 hex            : 91880b0e3017ba586b735fe7d04f1790f3c46b818a2151fb2def5f14dd2fd9c3
wif             : L26c3H6jEPVSqAr1usXUp9qtQJw6NHgApq6Ls4ncyqtsvcq2MwKH
 uncompressed   : 5JvNzA5vXDoKYJdw8SwwLHxUxaWvn9mDea6k1vRPCX7KLUVWa7W
public pair x   : 81821982719381104061777349269130419024493616650993589394553404347774
393168191
public pair y   : 58994218069605424278320703250689780154785099509277691723126325051200
459038290
 x as hex       : b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f
 y as hex       : 826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52
y parity        : even
key pair as sec : 02b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f
 uncompressed   : 04b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f
                    826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52
hash160         : 5d353a2ecdb262477172852d57a3f11de0c19286
 uncompressed   : e5bd3a7e6cb62b4c820e51200fb1c148d79e67da
Bitcoin address : 19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii
 uncompressed   : 1MwkRkogzBRMehBntgcq2aJhXCXStJTXHT
```

获取JSON信息：

```
$ ku P:foo -P -j
{
   "y_parity": "even",
   "public_pair_y_hex": "826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544d
e52",
   "private_key": "no",
   "parent_fingerprint": "00000000",
   "tree_depth": "0",
   "network": "Bitcoin",
   "btc_address_uncompressed": "1MwkRkogzBRMehBntgcq2aJhXCXStJTXHT",
   "key_pair_as_sec_uncompressed": "04b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b
4b4e8e67a0b0b13f826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52",
   "public_pair_x_hex": "b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b
13f",
   "wallet_key": "xpub661MyMwAqRbcFVF9ULcqLdsEa5WnCCugQAcgNd9iEMQ31tgH6u4DLQWoQayvtSVY
FvXz2vPPpbXE1qpjoUFidhjFj82pVShWu9curWmb2zy",
   "chain_code": "5eeb1023fd6dd1ae52a005ce0e73420821e1d90e08be980a85e9111fd7646bbc",
   "child_index": "0",
   "hash160_uncompressed": "e5bd3a7e6cb62b4c820e51200fb1c148d79e67da",
   "btc_address": "19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii",
   "fingerprint": "5d353a2e",
   "hash160": "5d353a2ecdb262477172852d57a3f11de0c19286",
   "input": "P:foo",
   "public_pair_x": "81821982719381104061777349269130419024493616650993589394553404347
774393168191",
   "public_pair_y": "58994218069605424278320703250689780154785099509277691723126325051
200459038290",
   "key_pair_as_sec": "02b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b
13f"
}
```

公共BIP32密钥：

```
$ ku -w -P P:foo
xpub661MyMwAqRbcFVF9ULcqLdsEa5WnCCugQAcgNd9iEMQ31tgH6u4DLQWoQayvtSVYFvXz2vPPpbXE1qpjoU
FidhjFj82pVShWu9curWmb2zy
```

生成子项：

```
$ ku -w -s3/2 P:foo
xprv9wTErTSkjVyJa1v4cUTFMFkWMe5eu8ErbQcs9xajnsUzCBT7ykHAwdrxvG3g3f6BFk7ms5hHBvmbdutNmy
g6iogWKxx6mefEw4M8EroLgKj
```

硬化子键：

```
$ ku -w -s3/2H P:foo
xprv9wTErTSu5AWGkDeUPmqBcbZWX1xq85ZNX9iQRQW9DXwygFp7iRGJo79dsVctcsCHsnZ3XU3DhsuaGZbDh8
iDkBN45k67UKsJUXM1JfRCdn1
```

WIF:

```
$ ku -W P:foo
L26c3H6jEPVSqAr1usXUp9qtQJw6NHgApq6Ls4ncyqtsvcq2MwKH
```

地址：

```
$ ku -a P:foo
19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii
```

生成一堆子项：

```
$ ku P:foo -s 0/0-5 -w
xprv9xWkBDfyBXmZjBG9EiXBpy67KK72fphUp9utJokEBFtjsjiuKUUDF5V3TU8U8cDzytqYnSekc8bYuJS8G3
bhXxKWB89Ggn2dzLcoJsuEdRK
xprv9xWkBDfyBXmZnzKf3bAGifK593gT7WJZPnYAmvc77gUQVej5QHckc5Adtwxa28ACmANi9XhCrRvtFqQcUx
t8rUgFz3souMiDdWxJDZnQxzx
xprv9xWkBDfyBXmZqdXA8y4SWqfBdy71gSW9sjx9JpCiJEiBwSMQyRxan6srXUPBtj3PTxQFkZJAiwoUpmvtrx
KZu4zfsnr3pqyy2vthpkwuoVq
xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8Esk
pzKL1Y8Gk9aX6QbryA5raK73p
xprv9xWkBDfyBXmZv2q3N66hhZ8DAcEnQDnXML1J62krJAcf7Xb1HJwuW2VMJQrCofY2jtFXdiEY8UsRNJfqK6
DAdyZXoMvtaLHyWQx3FS4A9zw
xprv9xWkBDfyBXmZw4jEYXUHYc9fT25k9irP87n2RqfJ5bqbjKdT84Mm7Wtc2xmzFuKg7iYf7XFHKkSsaYKWKJ
bR54bnyAD9GzjUYbAYTtN4ruo
```

生成相应的地址：

```
$ ku P:foo -s 0/0-5 -a
1MrjE78H1R1rqdFrmkjdHnPUdLCJALbv3x
1AnYyVEcuqeoVzH96zj1eYKwoWfwte2pxu
1GXr1kZfxE1FcK6ZRD5sqqqs5YfvuzA1Lb
116AXZc4bDVQrqmcinzu4aaPdrYqvuiBEK
1Cz2rTLjRM6pMnxPNrRKp9ZSvRtj5dDUML
1WstdwPnU6HEUPme1DQayN9nm6j7nDVEM
```

生成相应的WIF：

```
$ ku P:foo -s 0/0-5 -W
L5a4iE5k9gcJKGqX3FWmxzBYQc29PvZ6pgBaePLVqT5YByEnBomx
Kyjgne6GZwPGB6G6kJEhoPbmyjMP7D5d3zRbHVjwcq4iQXD9QqKQ
L4B3ygQxK6zH2NQGxLDee2H9v4Lvwg14cLJW7QwWPzCtKHdWMaQz
L2L2PZdorybUqkPjrmhem4Ax5EJvP7ijmxbNoQKnmTDMrqemY8UF
L2oD6vA4TUyqPF8QG4vhUFSgwCyuuvFZ3v8SKHYFDwkbM765Nrfd
KzChTbc3kZFxUSJ3Kt54cxsogeFAD9CCM4zGB22si8nfKcThQn8C
```

通过选择一个BIP32字符串（与子项0/3对应的字符串）来检查它是否工作：

```
$ ku -W xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj
5P7e8EskpzKL1Y8Gk9aX6QbryA5raK73p
L2L2PZdorybUqkPjrmhem4Ax5EJvP7ijmxbNoQKnmTDMrqemY8UF
$ ku -a xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj
5P7e8EskpzKL1Y8Gk9aX6QbryA5raK73p
116AXZc4bDVQrqmcinzu4aaPdrYqvuiBEK
```

是的，看起来很熟悉

从秘密指数：

```
$ ku 1


input            : 1
network          : Bitcoin
secret exponent : 1
 hex             : 1
wif              : KwDiBf89QgGbjEhKnhXJuH7LrciVrZi3qYjgd9M7rFU73sVHnoWn
 uncompressed    : 5HpHagT65TZzG1PH3CSu63k8DbpvD8s5ip4nEB3kEsreAnchuDf
public pair x    : 55066263022277343669578718895168534326250603453777594175500187360389
116729240
public pair y    : 32670510020758816978083085130507043184471273380659243275938904335757
337482424
 x as hex        : 79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
 y as hex        : 483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
y parity         : even
key pair as sec : 0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
 uncompressed    : 0479be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
                     483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
hash160          : 751e76e8199196d454941c45d1b3a323f1433bd6
 uncompressed    : 91b24bf9f5288532960ac687abb035127b1d28a5
Bitcoin address : 1BgGZ9tcN4rm9KBzDn7KprQz87SZ26SAMH
 uncompressed    : 1EHNa6Q4Jz2uvNExL497mE43ikXhwF6kZm
```

莱特币版本：

```
$ ku -nL 1


input            : 1
network          : Litecoin
secret exponent  : 1
 hex             : 1
wif              : T33ydQRKp4FCW5LCLLUB7deioUMoveiwekdwUwyfRDeGZm76aUjV
 uncompressed    : 6u823ozcyt2rjPH8Z2ErsSXJB5PPQwK7VVTwwN4mxLBFrao69XQ
public pair x    : 55066263022277343669578718895168534326250603453777594175500187360389116729240
public pair y    : 32670510020758816978083085130507043184471273380659243275938904335757337482424
 x as hex        : 79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
 y as hex        : 483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
y parity         : even
key pair as sec  : 0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
 uncompressed    : 0479be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
                     483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
hash160          : 751e76e8199196d454941c45d1b3a323f1433bd6
 uncompressed    : 91b24bf9f5288532960ac687abb035127b1d28a5
Litecoin address : LVuDpNCSSj6pQ7t9Pv6d6sUkLKoqDEVUnJ
 uncompressed    : LYWKqJhtPeGyBAw7WC8R3F7ovxtzAiubdM
```

狗狗币WIF:

```
$ ku -nD -W 1
QNcdLVw8fHkixm6NNyN6nVwxKek4u7qrioRbQmjxac5TVoTtZuot
```

从公共对（在Testnet）：

```
$ ku -nT 55066263022277343669578718895168534326250603453777594175500187360389116729240
,even


input                    : 55066263022277343669578718895168534326250603453777594175500187360389116729240
873603
                           89116729240,even
network                  : Bitcoin testnet
public pair x            : 55066263022277343669578718895168534326250603453777594175500187360389116729240
87360389116729240
public pair y            : 32670510020758816978083085130507043184471273380659243275938904335757337482424
04335757337482424
 x as hex                : 79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
1798
 y as hex                : 483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
d4b8
y parity                 : even
key pair as sec          : 0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
f81798
 uncompressed            : 0479be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
f81798
                           483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
10d4b8
hash160                  : 751e76e8199196d454941c45d1b3a323f1433bd6
 uncompressed            : 91b24bf9f5288532960ac687abb035127b1d28a5
Bitcoin testnet address : mrCDrCybB6J1vRfbwM5hemdJz73FwDBC8r
 uncompressed            : mtoKs9V381UAhUia3d7Vb9GNak8Qvmcsme
```

从hash160:

```
$ ku 751e76e8199196d454941c45d1b3a323f1433bd6


input           : 751e76e8199196d454941c45d1b3a323f1433bd6
network         : Bitcoin
hash160         : 751e76e8199196d454941c45d1b3a323f1433bd6
Bitcoin address : 1BgGZ9tcN4rm9KBzDn7KprQz87SZ26SAMH
```

作为狗狗币地址:

```
$ ku -nD 751e76e8199196d454941c45d1b3a323f1433bd6


input            : 751e76e8199196d454941c45d1b3a323f1433bd6
network          : Dogecoin
hash160          : 751e76e8199196d454941c45d1b3a323f1433bd6
Dogecoin address : DFpN6QqFfUm3gKNaxN6tNcab1FArL9cZLE
```

2.交易实用程序（TX）

命令行实用程序tx将以人类可读的形式显示交易，从pycoin的交易缓存或Web服务获取基础交易（当前支持blockchain.info和biteasy.com），合并交易，添加或删除输入或输出，以及签署交易。

以下是一些例子。

查看着名的“皮萨”交易：

```
$ tx 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
warning: consider setting environment variable PYCOIN_CACHE_DIR=~/.pycoin_cache to cac
he transactions fetched via web services
warning: no service providers found for get_tx; consider setting environment variable
PYCOIN_SERVICE_PROVIDERS=BLOCKR_IO:BLOCKCHAIN_INFO:BITEASY:BLOCKEXPLORER
usage: tx [-h] [-t TRANSACTION_VERSION] [-l LOCK_TIME] [-n NETWORK] [-a]
          [-i address] [-f path-to-private-keys] [-g GPG_ARGUMENT]
          [--remove-tx-in tx_in_index_to_delete]
          [--remove-tx-out tx_out_index_to_delete] [-F transaction-fee] [-u]
          [-b BITCOIND_URL] [-o path-to-output-file]
          argument [argument ...]
tx: error: can't find Tx with id 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f
67b17c40c2a
```

哎呀！ 我们没有设置Web服务。 现在我们来做：

```
$ PYCOIN_CACHE_DIR=~/.pycoin_cache
$ PYCOIN_SERVICE_PROVIDERS=BLOCKR_IO:BLOCKCHAIN_INFO:BITEASY:BLOCKEXPLORER
$ export PYCOIN_CACHE_DIR PYCOIN_SERVICE_PROVIDERS
```

这不是自动完成的，所以命令行工具不会泄漏潜在的关于您对第三方网站感兴趣的交易的私人信息。 如果您不在乎，可以将这些行放入.profile。

让我们再试一次：

```
$ tx 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
Version:  1  tx hash 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
 159 bytes
TxIn count: 1; TxOut count: 1
Lock time: 0 (valid anytime)
Input:
  0:                          (unknown) from 1e133f7de73ac7d074e2746a3d6717dfc99ecaa8e
9f9fade2cb8b0b20a5e0441:0
Output:
  0: 1CZDM6oTttND6WPdt3D6bydo7DYKzd9Qik receives 10000000.00000 mBTC
Total output 10000000.00000 mBTC
including unspents in hex dump since transaction not fully signed
010000000141045e0ab2b0b82cdefaf9e9a8ca9ec9df17673d6a74e274d0c73ae77d3f131e000000004a49
3046022100a7f26eda874931999c90f87f01ff1ffc76bcd058fe16137e0e63fdb6a35c2d78022100a61e91
99238eb73f07c8f209504c84b80f03e30ed8169edd44f80ed17ddf451901ffffffff010010a5d4e8000000
1976a9147ec1003336542cae8bded8909cdd6b5e48ba0ab688ac00000000


** can't validate transaction as source transactions missing
```

出现最后一行是为了验证交易的签名，您技术上需要源代码交易。 所以我们来添加-a来增加源信息的交易：

```
$ tx -a 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
warning: transaction fees recommendations casually calculated and estimates may be inc
orrect
warning: transaction fee lower than (casually calculated) expected value of 0.1 mBTC,
transaction might not propogate
Version:  1  tx hash 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
 159 bytes
TxIn count: 1; TxOut count: 1
Lock time: 0 (valid anytime)
Input:
  0: 17WFx2GQZUmh6Up2NDNCEDk3deYomdNCfk from 1e133f7de73ac7d074e2746a3d6717dfc99ecaa8e
9f9fade2cb8b0b20a5e0441:0 10000000.00000 mBTC  sig ok
Output:
  0: 1CZDM6oTttND6WPdt3D6bydo7DYKzd9Qik receives 10000000.00000 mBTC
Total input  10000000.00000 mBTC
Total output 10000000.00000 mBTC
Total fees        0.00000 mBTC


010000000141045e0ab2b0b82cdefaf9e9a8ca9ec9df17673d6a74e274d0c73ae77d3f131e000000004a49
3046022100a7f26eda874931999c90f87f01ff1ffc76bcd058fe16137e0e63fdb6a35c2d78022100a61e91
99238eb73f07c8f209504c84b80f03e30ed8169edd44f80ed17ddf451901ffffffff010010a5d4e8000000
1976a9147ec1003336542cae8bded8909cdd6b5e48ba0ab688ac00000000


all incoming transaction values validated
```

现在，我们来看一下特定地址（UTXO）的未使用输出。在块#1中，我们看到一个钱币交易到12c6DSiU4Rq3P4ZxziKxzrL5LmMBrzjrJX。让我们用fetch_unspent来查找这个地址中的所有钱币：

```
$ fetch_unspent 12c6DSiU4Rq3P4ZxziKxzrL5LmMBrzjrJX
a3a6f902a51a2cbebede144e48a88c05e608c2cce28024041a5b9874013a1e2a/0/76a914119b098e2e980
a229e139a9ed01a469e518e6f2688ac/333000
cea36d008badf5c7866894b191d3239de9582d89b6b452b596f1f1b76347f8cb/31/76a914119b098e2e98
0a229e139a9ed01a469e518e6f2688ac/10000
065ef6b1463f552f675622a5d1fd2c08d6324b4402049f68e767a719e2049e8d/86/76a914119b098e2e98
0a229e139a9ed01a469e518e6f2688ac/10000
a66dddd42f9f2491d3c336ce5527d45cc5c2163aaed3158f81dc054447f447a2/0/76a914119b098e2e980
a229e139a9ed01a469e518e6f2688ac/10000
ffd901679de65d4398de90cefe68d2c3ef073c41f7e8dbec2fb5cd75fe71dfe7/0/76a914119b098e2e980
a229e139a9ed01a469e518e6f2688ac/100
d658ab87cc053b8dbcfd4aa2717fd23cc3edfe90ec75351fadd6a0f7993b461d/5/76a914119b098e2e980
a229e139a9ed01a469e518e6f2688ac/911
36ebe0ca3237002acb12e1474a3859bde0ac84b419ec4ae373e63363ebef731c/1/76a914119b098e2e980
a229e139a9ed01a469e518e6f2688ac/100000
fd87f9adebb17f4ebb1673da76ff48ad29e64b7afa02fda0f2c14e43d220fe24/0/76a914119b098e2e980
a229e139a9ed01a469e518e6f2688ac/1
dfdf0b375a987f17056e5e919ee6eadd87dad36c09c4016d4a03cea15e5c05e3/1/76a914119b098e2e980
a229e139a9ed01a469e518e6f2688ac/1337
cb2679bfd0a557b2dc0d8a6116822f3fcbe281ca3f3e18d3855aa7ea378fa373/0/76a914119b098e2e980
a229e139a9ed01a469e518e6f2688ac/1337
d6be34ccf6edddc3cf69842dce99fe503bf632ba2c2adb0f95c63f6706ae0c52/1/76a914119b098e2e980
a229e139a9ed01a469e518e6f2688ac/2000000
    0e3e2357e806b6cdb1f70b54c3a3a17b6714ee1f0e68bebb44a74b1efd512098/0/410496b538e8535
19c726a2c91e61ec11600ae1390813a627c66fb8be7947be63c52da7589379515d4e0a604f8141781e6229
4721166bf621e73a82cbf2342c858eeac/5000000000
```

附录7、比特币浏览器命令

Bitcoin Explorer（bx）是一个命令行工具，提供了各种用于密钥管理和交易构建的命令。 它是libbitcoin比特币库的一部分。

```
Usage: bx COMMAND [--help]


Info: The bx commands are:


address-decode
address-embed
address-encode
address-validate
base16-decode
base16-encode
base58-decode
base58-encode
base58check-decode
base58check-encode
base64-decode
base64-encode
bitcoin160
bitcoin256
btc-to-satoshi
ec-add
ec-add-secrets
ec-multiply
ec-multiply-secrets
ec-new
ec-to-address
ec-to-public
ec-to-wif
fetch-balance
fetch-header
fetch-height
fetch-history
fetch-stealth
fetch-tx
fetch-tx-index
hd-new
hd-private
hd-public
hd-to-address
hd-to-ec
hd-to-public
hd-to-wif
help
```

```
input-set
input-sign
input-validate
message-sign
message-validate
mnemonic-decode
mnemonic-encode
ripemd160
satoshi-to-btc
script-decode
script-encode
script-to-address
seed
send-tx
send-tx-node
send-tx-p2p
settings
sha160
sha256
sha512
stealth-decode
stealth-encode
stealth-public
stealth-secret
stealth-shared
tx-decode
tx-encode
uri-decode
uri-encode
validate-tx
watch-address
wif-to-ec
wif-to-public
wrap-decode
wrap-encode
```

更多信息参见Bitcoin Explorer homepage 和 Bitcoin Explorer user documentation

bx命令使用示例

我们来看一些使用Bitcoin Explorer命令来测试密钥和地址的例子。

使用种子命令生成随机"种子"值，该种子命令使用操作系统的随机数生成器。将种子传递到ec-new命令以生成新的私钥。我们将标准输出保存到文件private_key中：

```
$ bx seed | bx ec-new > private_key
$ cat private_key
73096ed11ab9f1db6135857958ece7d73ea7c30862145bcc4bbc7649075de474
```

现在，使用ec-to-public命令从私钥生成公钥。 我们将private_key文件传递到标准输入并将命令的标准输出保存到新文件public_key中：

```
$ bx ec-to-public < private_key > public_key
$ cat public_key
02fca46a6006a62dfdd2dbb2149359d0d97a04f430f12a7626dd409256c12be500
```

我们可以使用ec-to-address命令将public_key重新格式化为一个地址。 我们将public_key传递给标准输入：

```
$ bx ec-to-address < public_key
17re1S4Q8ZHyCP8Kw7xQad1Lr6XUzWUnkG
```

以这种方式产生的密钥产生零型非确定性钱包。 这意味着每个密钥都是由一个独立的种子生成的。 Bitcoin Explorer命令也可以根据BIP-32确定性地生成密钥。 在这种情况下，从种子创建“主”键，然后确定性地扩展以产生一个子项的树，从而产生一个2类确定性钱包。

首先，我们使用seed和hd-new命令生成一个主密钥，该密钥将被用作导出密钥层次结构的基础：

```
$ bx seed > seed
$ cat seed
eb68ee9f3df6bd4441a9feadec179ff1


$ bx hd-new < seed > master
$ cat master
xprv9s21ZrQH143K2BEhMYpNQoUvAgiEjArAVaZaCTgsaGe6LsAnwubeiTcDzd23mAoyizm9cApe51gNfLMkBq
kYoWWMCRwzfuJk8RwF1SVEpAQ
```

我们现在使用hd-private命令在帐户中生成一个强化的“帐户”键和两个私钥序列：

```
$ bx hd-private --hard < master > account
$ cat account
xprv9vkDLt81dTKjwHB8fsVB5QK8cGnzveChzSrtCfvu3aMWvQaThp59ueufuyQ8Qi3qpjk4aKsbmbfxwcgS8P
YbgoR2NWHeLyvg4DhoEE68A1n


$ bx hd-private --index 0 < account
xprv9xHfb6w1vX9xgZyPNXVgAhPxSsEkeRcPHEUV5iJcVEsuUEACvR3NRY3fpGhcnBiDbvG4LgndirDsia1e9F
3DWPkX7Tp1V1u97HKG1FJwUpU


$ bx hd-private --index 1 < account
xprv9xHfb6w1vX9xjc8XbN4GN86jzNAZ6xHEqYxzbLB4fzHFd6VqCLPGRZFsdjsuMVERadbgDbziCRJru9n6tz
EWrASVpEdrZrFidt1RDfn4yA3
```

接下来，我们使用hd-public命令来生成两个公钥的相应序列：

```
$ bx hd-public --index 0 < account
xpub6BH1zcTuktiFu43rUZ2gXqLgzu5F3tLEeTQ5t6iE3aQtM2VMTxMcyLN9fYHiGhGpQe9QQYmqL2eYPFJ3ve
zHz5wzaSW4FiGrseNDR4LKqTy


$ bx hd-public --index 1 < account
xpub6BH1zcTuktiFx6CzhPbGjG3UYQ13WR16CmtbPiagEKpEVtpyjshWyMaMV1cn7nUPUkgQHPVXJVqsrA8xWb
GQDhohEcDFTEYMvYzwRD7Juf8
```

公钥也可以使用hd-to-public命令从其相应的私钥派生：

```
$ bx hd-private --index 0 < account | bx hd-to-public
xpub6BH1zcTuktiFu43rUZ2gXqLgzu5F3tLEeTQ5t6iE3aQtM2VMTxMcyLN9fYHiGhGpQe9QQYmqL2eYPFJ3ve
zHz5wzaSW4FiGrseNDR4LKqTy


$ bx hd-private --index 1 < account | bx hd-to-public
xpub6BH1zcTuktiFx6CzhPbGjG3UYQ13WR16CmtbPiagEKpEVtpyjshWyMaMV1cn7nUPUkgQHPVXJVqsrA8xWb
GQDhohEcDFTEYMvYzwRD7Juf8
```

我们可以在确定性链中产生几乎无限数量的密钥，全部来源于单个种子。 这种技术用于许多钱包应用程序中以生成可以使用单个种子值进行备份和恢复的密钥。 每次创建一个新的密钥时，这比将其所有随机生成的密钥备份在一起更容易。

可以使用助记符编码命令对种子进行编码：

```
$ bx hd-mnemonic < seed > words
adore repeat vision worst especially veil inch woman cast recall dwell appreciate
```

然后可以使用mnemonic-decode命令对种子进行解码：

```
$ bx mnemonic-decode < words
eb68ee9f3df6bd4441a9feadec179ff1
```

助记符编码可以使种子更容易记录甚至记住。

附录8、染色币

术语染色币是指一组使用比特币交易记录比特币以外的外部资产的创建，所有权和转让的类似技术。“外在”是指不直接存储在比特币区块链中的资产，而不是比特币本身，这是区块链固有的资产。

染色币用于追踪数字资产以及第三方持有的有形资产，并通过染色币进行所有权交易。数字资产染色币可以代表无形资产，如股票证书，许可证，虚拟财产（游戏物品），或任何形式的许可知识产权（商标，版权等）。有形资产的染色币可以代表商品（金，银，油），土地所有权，汽车，船只，飞机等的所有权证书。

这个术语来源于“着色”或标记比特币的名义数量的想法，例如单个satoshi，代表比特币价值本身之外的东西。作为一个类比，考虑在$ 1钞票上加上一个信息，说明“这是ACME的股票证书”或者“这个票据可以兑换成1盎司白银”，然后交易$ 1票据作为这个其他资产的所有权证书。第一次实施染色币，名为增强型基于订单的着色或EPOBC，将外部资产分配给1个满分输出。这样，这是一个真正的“染色币”。

在EPOBC之后，更新的染色币实现使用OP_RETURN脚本来存储关于外部资产的元数据。从某种意义上说，这些系统不是真正的染色币，因为没有硬币被“着色”。相反，使用OP_RETURN元数据的交易用于创建和跟踪将元数据关联到特定资产的外部数据存储的所有权。

今天染色币的两个最突出的实现是Colu的Open Assets和Coloured Coins。这两个系统对染色币使用不同的方法，并且不兼容。在一个系统中创建的染色币不能在其他系统中看到或使用。

1.使用染色币

染色币被创建，转移并通常在特殊的钱包中查看，可以解释附加在比特币交易上的染色币协议元数据。必须特别小心，避免在普通的比特币钱包中使用与染色币相关的钥匙，因为正常的钱包可能会破坏元数据。同样，不应将有颜色的硬币发送到由普通钱包管理的地址，而只发送到由可识别硬币的钱包管理的地址。 Colu和Open Assets系统都使用特殊的染色币地址来解决这个风险，并确保染色币不会被发送到不知道的钱包。

对于大多数通用区块链浏览器来说，染色币也是不可见的。相反，您必须使用染色币浏览器来解释染色币交易的元数据。

Am Open Assets兼容的钱包应用程序和区块链浏览器可以在以下位置找到：

coinprism：https://www.coinprism.info

一个Colu彩色钱币兼容的钱包应用程序和区块链浏览器可以在：

区块链资源管理器：http://coloredcoins.org/explorer/

Copay钱包插件：http://coloredcoins.org/colored-coins-copay-addon/

2.创造染色币

每个染色币的实现都有不同的方式来创建染色币，但是它们都提供了类似的功能。创建染色币资产的过程称为发行。初始交易，发行交易将资产注册在比特币区块链中，并创建用于参考资产的资产ID。一旦发布，资产可以使用转移交易在地址之间转移。

作为染色币发行的资产可以有多个属性。它们可以是可分割的或不可分割的，这意味着传输中资产的数量可以是整数（例如5）或者具有小数细分（例如4.321）。资产也可以有固定发行，即一次性发行一次，也可以重新发行，即初始发行后可以由原发行人发行新的资产单位。

最后，一些染色币能够发行股息，允许比特币的付款给所有权比例的染色币资产的所有者。